CCP 2011. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

Partie I. Logique et calcul des propositions.

1) Règles de politesse appliquées à la première discussion :
$(A_1 \Longrightarrow B_1)(B_1 \Longrightarrow C_1) \equiv (\overline{A_1} + B_1)(\overline{B_1} + C_1) \equiv \overline{A_1}\ \overline{B_1} + \overline{A_1}\ C_1 + B_1C_1$ □

2) $A_1 \equiv D + G \qquad B_1 \equiv \overline{D} \qquad C_1 \equiv GP$ □

3) Il vient $\overline{A_1}\ \overline{B_1} \equiv \overline{D}\ \overline{G}D \equiv$ faux $\qquad \overline{A_1}C_1 \equiv \overline{D}\ \overline{G}GP \equiv$ faux $\qquad B_1C_1 \equiv \overline{D}\ GP$
Donc un kjalt possède des griffes et des pinces mais pas de dard. □

4) Règles de politesse appliquées à la seconde discussion :
$(A_2 \Longrightarrow B_2)(B_2 \Longrightarrow A_3)(A_2 \Longleftrightarrow A_3)$ □

5) $A_2 \equiv M\overline{J} \qquad B_2 \equiv \overline{V} \qquad A_3 \equiv (V \Longrightarrow J) \equiv \overline{V} + J$ □

6)

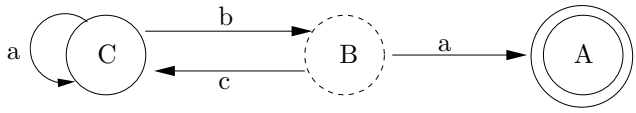
J	M	V	$A_2 \equiv \overline{J}M$	$B_2 \equiv \overline{V}$	$A_3 \equiv J + \overline{V}$	$A_2 \Longrightarrow B_2$	$B_2 \Longrightarrow A_3$	$A_2 \Longleftrightarrow A_3$
0	0	0	0	1	1	1	1	0
0	0	1	0	0	0	1	1	1
0	1	0	1	1	1	1	1	1
0	1	1	1	0	0	0	1	0
1	0	0	0	1	1	1	1	0
1	0	1	0	0	1	1	1	0
1	1	0	0	1	1	1	1	0
1	1	1	0	0	1	1	1	0

Donc un lyop peut être soit vert mais ni jaune ni mauve soit mauve mais ni jaune ni vert. □

Partie II. Automates et langages.

1) Le langage reconnu par l'automate $\mathcal{E}$ est défini par l'expression régulière $L = ab(cb + a)^*$. □

2)



3) Le langage reconnu par l'automate $\mathcal{E}^{-1}$ est défini par l'expression régulière $(a + bc)^*ba$.
Il s'agit donc du langage miroir $\widetilde{L}$ de L. □

4) Démonstration immédiate par récurrence sur la longueur de m. □

5) On établit la propriété demandée par récurrence sur la longueur n du mot m.
Si $n = 1$ il s'agit de la définition même de la fonction de transition δ^{-1}.
Supposons la proprité vraie jusqu'au rang $n \geqslant 1$ et soit $m = u.x$ un mot de longueur $n + 1$.
D'après la question précédente on a : $\Big(d \in \delta^*(o, m)\Big) \Longleftrightarrow \Big(\exists q \in Q \setminus (q \in \delta^*(o, u)) \wedge (d \in \delta(q, x))\Big)$
Or par hypothèse de récurrence $\Big(q \in \delta^*(o, u)\Big) \Longleftrightarrow \Big(o \in \delta^{-1*}(q, u^{-1})\Big)$ et $\Big(d \in \delta(q, x)\Big) \Longleftrightarrow \Big(q \in \delta^{-1}(d, x)\Big)$
Ainsi $\Big(d \in \delta^*(o, m)\Big) \Longleftrightarrow \Big(\exists q \in Q \setminus (q \in \delta^{-1}(d, x)) \wedge (o \in \delta^{-1*}(q, u^{-1}))\Big)$
$\Longleftrightarrow \Big(o \in \delta^{-1*}(d, xu^{-1})\Big)$ par définition de la fonction de transition généralisée δ^{-1*}
$\Longleftrightarrow \Big(o \in \delta^{-1*}(d, m^{-1})\Big)$
Ce qui établit la propriété au rang $n + 1$ et donc d'une manière générale. □

6) Il en résulte que si L est un langage rationnel reconnu par l'automate $\mathcal{A}$ alors le langage reconnu par $\mathcal{A}^{-1}$ est le langage miroir $\widetilde{L}$ de L. □

Partie III. Algorithmique et programmation.

Arbres binaires d'entiers.

1) $|\varnothing| = -1 \ \wedge \ |a| = \max\big(|\mathcal{G}(a), \mathcal{D}(a)\big) + 1$ □

2) Un arbre a contenant un ensemble X d'entiers de cardinal $n \geqslant 1$ dont la profondeur est maximale est filiforme (sa profondeur est $n-1$) et celui dont la profondeur est minimale est quasi-complet *i.e.* tel que tous les niveaux sont complets sauf le plus profond et éventuellement celui d'avant en d'autres termes toutes les feuilles sont à profondeur $|a|$ ou $|a-1|$. □

3) Immédiat par récurrence sur p. □

4) Il en résulte que $n = 1 + 2 + 2^2 + \ldots + 2^p = 2^{p+1} - 1$ □

5) Donc $p = \log_2(n+1) - 1$ □

Arbres binaires de recherche..

6) Dans l'exemple proposé on a successivement les appels suivants :
`aux1 [2; 1; 3] Vide` qui appelle
`ajouter 2 Vide` qui renvoie `a1=Noeud(Vide,2,Vide)`
  `aux1 [1; 3] a1` qui appelle
  `ajouter 1 a1` qui renvoie `a2=Noeud(Noeud(Vide,1,Vide),2,Vide)`
    `aux1 [3] a2` qui appelle
    `ajouter 3 a2` qui renvoie `a3=Noeud(Noeud(Vide,1,Vide),2,Noeud(Vide,3,Vide))`

puis `aux2 a3` qui
1/ appelle `aux2 Noeud(Vide,1,Vide)` et `aux2 Noeud(Vide,3,Vide)` qui renvoient respectivement `[1]` et `[3]`
2/ renvoie `[1]@(2::[3])=[1; 2; 3]`

7) Il est immédiat par récurrence sur la taille de p que `ajouter e p` où p est un ABR et e un entier renvoie l'ABR obtenu à partir de p en ajoutant l'étiquette e (avec répétition si e figurait déjà dans p). □

8) Il en résulte par récurrence sur la longueur de s que `aux1 s a` où s est une séquence et a un ABR renvoie l'ABR obtenu en ajoutant successivement les valeurs de s dans a.
Donc `aux1 s Vide` renvoie un ABR qui a pour étiquettes les éléments de la séquence s avec les mêmes éventuelles répétitions.
Par récurrence sur la taille de l'arbre a on prouve immédiatement que `aux2 a` où a est un AB renvoie la séquence des étiquettes obtenues en parcourant a en profondeur depuis la branche la plus à gauche en partant du bas. Donc si a est un ABR elle renvoie la séquence classée par ordre croissant des étiquettes de a.

Il en résulte que `trier s` renvoie la séquence `s` triée par ordre croissant au sens large. □

9) • La fonction `ajouter v a` est récursive sur l'ensemble des $\mathbb{N} \times ABR$ muni de graduation d(v,a) =taille(a) si v n'est pas la racine de a et $d(v,a) = 0$ sinon.
Elle se termine donc bien sur un cas de base : soit $(v, \varnothing)$ si v n'est pas une étiquette de a soit (v,r) avec v racine de r. □

Le nombre d'appels récursifs est égal à la longueur de la branche suivie jusqu'à trouver v s'il figure déjà où l'arbre vide sinon.

• La fonction `aux1 s a` est récursive sur l'ensemble des listes chaînées muni de la graduation longueur (fonction de descente usuelle $n \longmapsto n-1$). Elle se termine donc bien sur le cas de base de la liste vide. □

Le nombre d'appels récursifs de `aux 1 s a` est égal à la longueur de la liste chaînée et chaque appel déclenche un appel à `ajouter`.

• La fonction `aux2 a` est doublement récursive sur l'ensemble des ABR muni de la graduation taille. Elle se termine donc bien sur le cas de base de l'arbre vide. □

Le nombre d'appels récursifs de `aux 2` est égal au nombre de noeuds de l'arbre a car tous les noeuds sont visités et cela ne déclenche aucun appel aux deux autres fonctions.

10) Soit s de longueur n.
La complexité de `aux1 s Vide` est égale à la somme des longueurs de la branche visitée de l'arbre courant pour chaque ajout de la tête de la séquence courante.
• Cette complexité est maximale si l'arbre construit est à chaque instant filiforme et si la valeur à ajouter vient à la fin de cette branche. Cela se produit si la séquence s est constituée d'une suite croissante ou décroissante d'entiers. Auquel cas la complexité en appels récursifs de `ajouter` est égale à $+1 + 2 + \ldots + (n-1) = \theta(n^2)$

• Elle est minimale (cela resterait à démontrer rigoureusement) si l'arbre construit est à chaque instant quasi-complet auquel cas la complexité est de l'ordre de (Cf question 5) $\log_2 2+\log_2 3+\ldots+\log_2(n-1) = \log_2((n-1)!) \sim n\log_2 n$
Cela se produit évidemment si s est la séquence obtenue en parcourant en largeur un ABR complet !
Par exemple $s = [8;4;12;2;6;10;12;14;1;3;5;7;9;11;13;15]$

Dans tous les cas la complexité de la fonction `aux2` est de n appels récursifs à elle-même.
Remarquons que cette estimation de complexité ne prend pas en compte le coût des concaténations dans la fonction `aux2`.

• En conclusion on peut retenir un nombre d'appels récursifs en $\Theta(n^2)$ dans le pire des cas d'une séquence déjà triée et en $\Theta(n\log_2(n))$ dans le meilleur des cas d'une séquence correspondant au parcours en largeur d'un ABR quasi-complet. □

Représentation de systèmes creux.

11) Démontrons la propriété demandée par récurrence sur p, la vérification étant immédiate pour $p = 0$.
Supposons que les 2^p noeuds de profondeur p aient des numéros décrivant l'ensemble $I_p = [\![2^2, 2^{p+1}[\![$ (ce qui représente bien 2^p entiers différents deux à deux).
Quand n parcourt I_p, $n+2^p$ décrit exactement $I_{p,g} = [\![2^{p+1}, 2^{p+1}+2^p[\![$ et $n+2^{p+1}$ décrit $I_{p,d} = [\![2^p+2^{p+1}, 2^{p+2}[\![$.
Or ces deux ensembles sont disjoints et d'union $[\![2^{p+1}, 2^{p+2}[\![= I_{p+1}$ ce qui établit l'hérédité de la propriété. □

12) Les numéros des noeuds de même profondeur sont deux à deux différents comme on vient de le voir et en outre si $p \neq q$ alors $I_p \cap I_q = \varnothing$ d'où l'unicité de la numérotation. □

13) Raisonnons par récurrence sur p, la propriété étant bien vérifiée pour $p = 0$.
Supposons la propriété satisfaite jusqu'au rang $p-1$ avec $p \geqslant 1$ et envisageons le noeud de numéro n d'occurence $(c_1, c_2, \ldots, c_p)$.
Il s'agit d'un fils du noeud d'occurence $(c_1, c_2, \ldots, c_{p-1})$ qui, par hypothèse de récurrence, a pour numéro
$m = 2^{p-1} + \sum_{i=0}^{p-2} c_{i+1} \times 2^i$.
Si $c_p = 0$ il s'agit du fils gauche donc :
$n = m + 2^{p-1} = 2^{p-1} + \sum_{i=0}^{p-2} c_{i+1} \times 2^i + 2^{p-1} = 2^p + \sum_{i=0}^{p-2} c_{i+1} \times 2^i = 2^p + \sum_{i=0}^{p-1} c_{i+1} \times 2^i$ car $c_p = 0$
Si $c_p = 1$ il s'agit du fils droit donc :
$n = m + 2^p = 2^{p-1} + \sum_{i=0}^{p-2} c_{i+1} \times 2^i + 2^p = 2^p + \sum_{i=0}^{p-1} c_{i+1} \times 2^i$ car $c_p = 1$
Dans les deux cas la propriété est donc bien satisfaite au rang p ce qui établit son hérédité. □

En d'autres termes $\overline{1c_pc_{p-1}\ldots c_2c_1}$ est l'écriture de n en base 2.

14) Donc c_i est le reste de la division par deux de la division entière de n par 2^{i-1} □

15) On écrit facilement :

```
let rec taille a = match a with
  | Vide -> 0
  | Fourche (fg,fd) -> taille fg + taille fd
  | Noeud(fg,a,fd) -> taille fg + taille fd + 1
;;
taille : arbre -> int = <fun>
```

16) On commence par écrire 4 fonctions qui vont faciliter la lecture de la fonction `bornes` et qui seront également utiles dans la suite.

La fonction `num_g` de sorte que si `n` est le numéro d'un noeud dans un arbre `fg`, `num_g n` renvoie le numéro de ce noeud dans un arbre `Fourche(fg,_)` ou `Noeud(fg,_,_)`.
Idem avec la fonction `num_d` pour le fils droit.

```
let num_g n = if (n = 0) then 0 else 2 * n
and num_d n = if (n = 0) then 0 else 2 * n + 1;;
```

La fonction `num_mini` de sorte que si `mg` et `md` sont le plus petit des indices des étiquettes des fils gauche et droit d'une fourche (mais pas d'un noeud bien sûr !), `num_mini mg md` renvoie le plus petit indice des étiquettes dans cette fourche.
La fonction `num_maxi` de sorte que si `Mg` et `Md` sont le plus grand des indices des étiquettes des fils gauche et droit d'une fourche ou d'un noeud, `num_mini mg md` renvoie le plus grand indice des étiquettes dans cette fourche ou ce noeud.

```
let num_mini mg md = if (mg = 0) then num_d md
                     else if (md = 0) then num_g mg
                     else if (mg < md) then num_g mg
                     else num_d md
and num_maxi Mg Md = if (max Mg Md = 0) then 0
                     else if (Mg > Md) then num_g Mg
                     else num_d Md
;;
num_mini : int -> int -> int = <fun>
num_maxi : int -> int -> int = <fun>
```

On écrit alors immédiatement la fonction demandée :

```
let rec bornes a = match a with
  | Vide -> (0,0)
  | Fourche(fg,fd) -> let (mg,Mg) = bornes fg and (md,Md)= bornes fd in
                        (num_mini mg md, num_maxi Mg Md)
  | Noeud(fg,_,fd) -> let (mg,Mg) = bornes fg and (md,Md)= bornes fd in
                        (1, max 1 (num_maxi Mg Md))
;;
bornes : arbre -> int * int = <fun>
```

17) Écriture récursive immédiate en remarquant que les noeuds du fils gauche (resp. droit) d'un arbre non vide ont tous des numéros pairs (resp. impair).

```
let rec remplacer i e a = match a with
  | Vide  -> failwith " Remplacement impossible"
  | Fourche (fg,fd) -> if (i = 1) then failwith "Remplacement impossible"
                       else if (i mod 2 = 0) then Fourche(remplacer (i/2) e fg,fd)
                       else Fourche(fg,remplacer (i/2) e fd)
  | Noeud (fg,v,fd) -> if (i = 1) then Noeud(fg,e,fd)
                       else if (i mod 2 = 0) then Noeud(remplacer (i/2) e fg,v,fd)
                       else Noeud(fg,v,remplacer (i/2) e fd)
;;
remplacer : int -> float -> arbre -> arbre = <fun>
```

18) $VBF(v) = \Big((\mathcal{V}_{min}(v), \mathcal{V}_{max}(v)) = \text{bornes}(\mathcal{V}_a(v))\Big) \wedge \Big(\forall x \in \mathcal{N}(\mathcal{V}_a(v)) \;\; \mathcal{E}(x) \neq \mathcal{V}_d(v)\Big) \wedge \Big(\text{taille}(\mathcal{V}_a(v)) = \mathcal{V}_t(v)\Big)$
$\wedge \Big(\forall x \in \mathcal{F}(\mathcal{V}_a(v)) \;\; \big(\mathcal{G}(x) \neq \varnothing\big) \vee \big(\mathcal{D}(x) \neq \varnothing\big)\Big)$

19) Une simple adaptation de la fonction `bornes` permet d'écrire une fonction `test` de sorte que `test v a` renvoie (avec un seul parcours de l'arbre partiel `a`) le quadruplet $(0,0,0,false)$ si l'arbre contient une fourche dont les deux fils sont vides ou une étiquette égale à `v` et sinon $(m, M, t, true)$ où t est le nombre d'étiquettes de l'arbre, m (resp. M) le plus petit (resp. grand) indice d'une étiquette.

```
let rec test v a = match a with
  | Vide -> (0,0,0,true)
  | Fourche (fg,fd) ->
      if (fg = Vide) && (fd = Vide) then (0,0,0,false)
      else let (mg,Mg,tg,bg) = test v fg and (md,Md,td,bd) = test v fd in
        if (not bg) || (not bd) then (0,0,0,false)
        else (num_mini mg md, num_maxi Mg Md, tg + td, true)
  | Noeud(fg,e,fd) ->
      if (e = v) then (0,0,0,false)
      else let (mg,Mg,tg,bg) = test v fg and (md,Md,td,bd) = test v fd in
        if (not bg) || (not bd) then (0,0,0,false)
        else (1, max 1 (num_maxi Mg Md), tg + td + 1, true)
;;
test : float -> arbre -> int * int * int * bool = <fun>
```

La fonction `valider` est alors immédiate :

```
let valider (imin,imax,t,v,a) = let (m,M,ta,b)= test v a in
  b && (m = imin) && (M = imax) && (ta = t)
;;
valider : int * int * int * float * arbre -> bool = <fun>
```

20) On commence par écrire une fonction `indice` de sorte que `indice n a` où n est un entier strictement positif et a un arbre binaire partiel renvoie le sous-arbre d'indice n de a s'il existe et une exception sinon.

```
let rec indice n a = if (n = 1) then a else match a with
  | Vide -> failwith "Indice non valide"
  | Fourche(fg,fd) -> indice (n / 2) ( if (n mod 2 = 0) then fg else fd)
  | Noeud(fg,_,fd) -> indice (n / 2) ( if (n mod 2 = 0) then fg else fd)
;;
indice : int -> arbre -> arbre = <fun>
```

La fonction `lire` est alors immédiate en notant que comme le vecteur est bien formé si n est l'indice d'un sous-arbre vide cet indice n'est pas un indice valide du vecteur.

```
let lire n (imin,imax,t,v,a) = match (indice n a) with
              | Vide           -> failwith "Indice incorrect"
              | Fourche(fg,fd) -> v
              | Noeud(_,e,_)   -> e
;;
lire : int -> 'a * 'b * 'c * float * arbre -> float = <fun>
```

Remarque : si on sait a prori que n est un indice valide du vecteur, on peut améliorer en évitant le parcours de l'arbre si n n'est pas dans l'intervalle des valeurs autres que la valeur par défaut :

```
let lire_bis n (imin,imax,t,v,a) =
  if (n < imin) || (n > imax) then v
  else match (indice n a) with
    | Fourche(fg,fd) -> v
    | Noeud(_,e,_)   -> e
;;
lire_bis : int -> int * int * 'a * float * arbre -> float = <fun>
```

21) On commence par modifier facilement la fonction `remplacer` de la question 17 par `remplacer_bis` de sorte que `remplacer_bis n e a` renvoie :
- une exception si n n'est pas un indice valide;
- si n est l'indice de `Fourche(fg,fd)` le couple $(aa, 1)$ où aa est l'arbre obtenu à partir de a en remplaçant `Fourche(fg,fd)` par `Noeud(fg,e,fd)`;
- si n est l'indice de `Noeud(fg,x,fd)` le couple $(aa, 0)$ où aa est l'arbre obtenu à partir de a en remplaçant `Noeud(fg,x,fd)` par `Noeud(fg,e,fd)`.

```
let rec remplacer_bis n e a = match a with
  | Vide  -> failwith " Remplacement impossible !"
  | Fourche (fg,fd) -> if (n = 1) then (Noeud(fg,e,fd),1)
                       else if (n mod 2 = 0)
                         then let (aa,i) = remplacer_bis (n/2) e fg in (Fourche(aa,fd),i)
                         else let (aa,i) = remplacer_bis (n/2) e fd in (Fourche(fg,aa),i)
  | Noeud (fg,x,fd) -> if (n = 1) then (Noeud(fg,e,fd),0)
                       else if (n mod 2 = 0)
                         then let (aa,i) = remplacer_bis (n/2) e fg in (Noeud(aa,v,fd),i)
                         else let (aa,i) = remplacer_bis (n/2) e fd in (Noeud(fg,v,aa),i)
;;
remplacer_bis : int -> float -> arbre -> arbre * int = <fun>
```

On écrit alors immédiatement la fonction `ecrire` :

```
let ecrire n e (imin,imax,t,v,a) = let (aa,i) = remplacer_bis n e a in
  if (i = 0) then (imin,imax,t,v,aa)
  else ((min n imin),(max n imax),t+1,v,aa)
;;
ecrire : int -> float -> int * int * int * 'a * arbre -> int * int * int * 'a * arbre = <fun>
```

22) La question est imprécise. Si les deux vecteurs creux ont la même valeur par défaut v et si on suppose que la somme de deux valeurs (non par défaut) e_1 et e_2 de même indice n'est jamais égale à v, la fonction est facile à écrire.
On écarte ces hypothèses simplificatrices. On désigne par v_1 et v_2 les valeurs par défaut de sorte que $v = v_1 + v_2$ est la valeur par défaut de la somme. On désigne par a_1 et a_2 les deux arbres partiels (bien formés) associés aux deux vecteurs creux et on va écrire une fonction `ajout v a1 a2` qui va renvoyer (avec un seul parcours de deux

arbres) le vecteur creux somme *i.e.* le quadruplet `(m,M,t,v,a)` où a est l'arbre partiel (bien formé) associé à la somme des deux vecteurs creux, t le nombre de valeurs différentes de la valeur par défaut v, m l'indice minimum d'une telle valeur et M l'indice maximum.
En effet les valeurs de la taille et des indices minimum et maximum des deux vecteurs creux arguments n'influent pas sur le résultat dans ce cas général.

Pour traiter le cas de base où l'un des deux arbres courants est vide, on commence par modifier facilement la fonction `bornes` de la question 16 en une fonction `bornes_taille` qui renvoie en un seul parcours les bornes et la taille :

```
let rec bornes_taille a = match a with
  | Vide -> (0,0,0)
  | Fourche(fg,fd) -> let (mg,Mg,tg) = bornes_taille fg and (md,Md,td)= bornes_taille fd in
                        (num_mini mg md, num_maxi Mg Md, tg + td)
  | Noeud(fg,_,fd) -> let (mg,Mg,tg) = bornes_taille fg and (md,Md,td)= bornes_taille fd in
                        (1, max 1 (num_maxi Mg Md), tg + td + 1)
;;
bornes_taille : arbre -> int * int * int = <fun>
```

Puis on écrit la fonction `ajout v a1 a2` :

```
let rec ajout v a1 a2 = match (a1,a2) with
  | (Vide,a2) -> let (m,M,t) = bornes_taille a2 in (m,M,t,a2)
  | (a1,Vide) -> let (m,M,t) = bornes_taille a1 in (m,M,t,a1)
  | (Fourche(fg1,fd1),Fourche(fg2,fd2)) ->
      let (mg,Mg,tg,fg) = ajout v fg1 fg2
      and (md,Md,td,fd) = ajout v fd1 fd2 in
        (num_mini mg md,num_maxi Mg Md,tg + td,Fourche(fg,fd))
  | (Fourche(fg1,fd1),Noeud(fg2,e,fd2)) ->
      let (mg,Mg,tg,fg) = ajout v fg1 fg2
      and (md,Md,td,fd) = ajout v fd1 fd2 in
        (1,max (num_maxi Mg Md) 1,tg + td + 1,Noeud(fg,e,fd))
  | (Noeud(fg1,e,fd1),Fourche(fg2,fd2)) ->
      let (mg,Mg,tg,fg) = ajout v fg1 fg2
      and (md,Md,td,fd) = ajout v fd1 fd2 in
        (1,max (num_maxi Mg Md) 1,tg + td + 1,Noeud(fg,e,fd))
  | (Noeud(fg1,e1,fd1),Noeud(fg2,e2,fd2)) ->
      let (mg,Mg,tg,fg) = ajout v fg1 fg2
      and (md,Md,td,fd) = ajout v fd1 fd2 in
        if (e1 +. e2 = v) then (num_mini mg md,num_maxi Mg Md,tg + td,Fourche(fg,fd))
        else (1,max (num_maxi Mg Md) 1,tg + td + 1,Noeud(fg,e1 +. e2,fd));;
ajout : float -> arbre -> arbre -> int * int * int * arbre = <fun>
```

On peut alors évidemment écrire :

```
let ajouter (_,_,_,v1,a1) (_,_,_,v2,a2) =
  let v = v1 +. v2 in
    let (m,M,t,a) = ajout v a1 a2 in
      (m,M,t,v,a)
;;
ajouter :
 'a * 'b * 'c * float * arbre ->
 'd * 'e * 'f * float * arbre -> int * int * int * float * arbre = <fun>
```

FIN