CCP 2012. Option Informatique.

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

Partie I. Logique et calcul des propositions.

1) Ce nombre minimum est évidemment de 2 (atteint lorsqu'on peut attribuer une valeur de vérité à chacune des deux déclarations de l'orateur). □

2) $A_V \equiv A_1A_2\ldots A_n \quad A_M \equiv \overline{A_1}\ \overline{A_2}\ldots\overline{A_n} \quad A_{CV} \equiv A_1\overline{A_2}\ldots(-1)^{n+1}A_n \quad A_{CM} =\equiv \overline{A_1}A_2\ldots(-1)^nA_n$
avec la convention $(-1)^kA_i = A_i$ si i est pair et $\overline{A_i}$ sinon. □

3) $A_1 \equiv R\overline{B}; \quad A_2 \equiv R+V; \quad A_3 \equiv (RV) \Longrightarrow B \equiv \overline{RV}+B \equiv \overline{R}+\overline{V}+B$ □

Notons aussi que $\overline{A_1} \equiv \overline{R}+B; \quad \overline{A_2} \equiv \overline{R}\ \overline{V}; \quad \overline{A_3} \equiv RV\overline{B}$

4) $A_V \equiv (R\overline{B})(R+V)(\overline{R}+\overline{V}+B) \equiv (R\overline{B})(R\overline{V}+RB+V\overline{R}+VB) \equiv R\overline{B}\ \overline{V}$
$A_M \equiv (\overline{R}+B)(\overline{R}\ \overline{V})(RV\overline{B}) \equiv (\overline{R}+B)faux \equiv faux$
$A_{CV} \equiv (R\overline{B})(\overline{R}\ \overline{V})(\overline{R}+\overline{V}+B) \equiv faux(\overline{R}+\overline{V}+B) \equiv faux$
$A_{CM} \equiv (\overline{R}+B)(R+V)(RV\overline{B}) \equiv (\overline{R}+B)(RV\overline{B}) \equiv faux$ □

5) Il en découle que l'orateur A est de type véridique et qu'il n'aime que le rouge. □

6) $G_1 \equiv (C \Longrightarrow L) \equiv \overline{C}+L; \quad H_1 \equiv C+T; \quad I_1 \equiv L; \quad I_2 \equiv \overline{T}$ □

7) Les informations sur la nature des orateurs sont représentées par la formule $I \equiv \overline{G_1}H_1(I_1\overline{I_2}+\overline{I_1}I_2)$ □

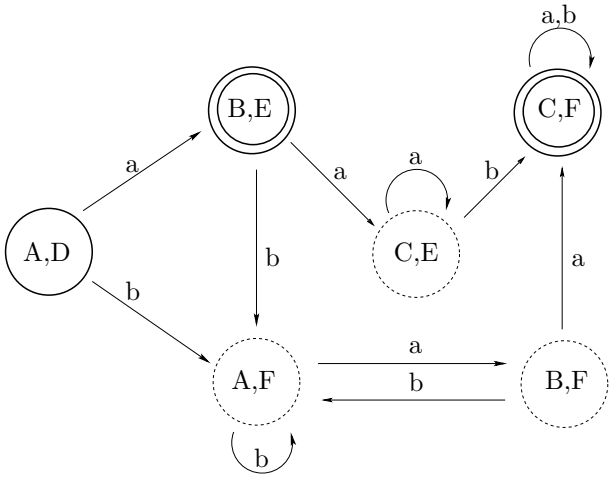
8) On a $\overline{G_1}H_1 \equiv C\overline{L}(C+T) \equiv C\overline{L}$ et $I_1\overline{I_2}+\overline{I_1}I_2 \equiv LT+\overline{L}\ \overline{T}$
De sorte que $I \equiv C\overline{L}(LT+\overline{L}\ \overline{T}) \equiv C\overline{L}\ \overline{T}$

Ainsi seul le cercle est visible et I commence par mentir □

Partie II. Automates et langages.

1) Le langage régulier (théorème de Kleene) reconnu par $\mathcal{E}_1$ peut être décrit par l'expression régulière $L_1 = (ab+b)^*a^2(a+b)^*$.
Celui reconnu par $\mathcal{E}_2$ peut évidemment être décrit par $L_2 = aa^*$. □

2)



3) Le langage reconnu par $\mathcal{E}_1 \oplus \mathcal{E}_2$ est décrit par l'expression régulière $a + \Big((ab+b)(ab+b)^*a^2 + a^2a^*b\Big)(a+b)^*$ □

4) Évident par définition même de la composition de deux automates finis complets déterministes. □

5) Démonstration immédiate par récurrence sur $n = |m|$. □

6) Résulte immédiatement de la question précédente et de la définition des états finaux de $\mathcal{A}_1 \oplus \mathcal{A}_2$. □

7) Il en résulte que $L_1 \oplus L_2$ est l'ensemble des mots appartenant à l'un des langages mais pas à l'autre. □

On peut le vérifier sur les expressions régulières :
1/ Si $m = a$ alors $m \in L_2 = aa^*$ mais $m \notin L_1$ car tout mot de L_1 contient le sous-mot a^2;
2/ Si $m \in \Big((ab+b)(ab+b)^*a^2 + a^2a^*b\Big)(a+b)^*$ alors $m \in (ab+b)^*a^2(a+b)^* = L_1$ clairement mais $m \notin L_2$ car m contient la lettre b.

Partie III. Algorithmique et programmation en CaML.

1/ Séquence croissante d'entiers

1)
```
let rec ajoutSequence v s = match s with
  | [] -> [v]
  | a::suite -> if (v <= a) then v::s else a::(ajoutSequence v suite)
;;
ajoutSequence : 'a -> 'a list -> 'a list = <fun>
```

2) La complexité est au mieux à temps constant lorsque l'élément v à ajouter est inférieur au premier de la séquence s et au pire linéaire lorsque v est plus grand que tous les éléments de s car alors l'équation de complexité est $T(n) = T(n-1) + \alpha$. □

3) On commence par écrire une fonction `partage` de sorte que `partage p s` renvoie `(s1,v,s2)` avec `s=s1@(v::s2)` où `v` est l'élément de `s` venant en position p.
Les arguments sont supposés de sorte que `s` est non vide et $1 \leqslant p \leqslant n$ avec n la longueur de `s`.

```
let rec partage p s = match p with
  | 1 -> ([],hd s, tl s)
  | _ -> let (a::suite) = s in
           let (s1,v,s2) = partage (p-1) suite in
             (a::s1,v,s2)
;;
partage : int -> 'a list -> 'a list * 'a * 'a list = <fun>
```

D'où la fonction demandée s'appliquant à une séquence non vide de longueur impaire :

```
let scissionSequence s =
  let n = length s in partage ((n+1)/2) s
;;
scissionSequence : 'a list -> 'a list * 'a * 'a list = <fun>
```

2/ Arbres binaires de recherche d'entiers

4) `eliminer 2 exemple` empile dans (1) `eliminer 2 Noeud(Noeud(Vide,1,Vide),2,Vide)`
qui empile dans (2) `eliminer 2 Noeud(Vide,1,Vide)`
qui empile dans (3) `eliminer 2 Vide` qui renvoie Vide (cas de base de `eliminer`)
Alors (2) renvoie `Noeud(Vide,1,Vide)`
puis (1) appelle `aux Noeud(Vide,1,Vide)` qui renvoie `(1,Vide)` (cas de base de `aux`)
et (1) renvoie finalement `Noeud(Vide,1,Noeud(Vide,3,Vide))` □

5) 6) • Commençons par étudier la fonction `aux` et prouver que :
(1) elle se termine dans tous les cas;
(2) sa complexité est égale à la longueur de la branche la plus à droite de son paramètre `a`;
(3) si `a=Noeug(g,v,d)` est non vide alors `(w,b)=aux a` est tel que :
$ABR(b)$, $C(a) = C(b) \cup \{w\}$ et $x \leqslant w$ pour $x \in C(b)$.

Notons que la propriété (3) revient à dire que `w` est un élément maximal de `a`.
Remarquons surtout que cela revient à dire que `Noeud(b,w,Vide)` est une réorganisation de a en un ABR dont la racine est un élément maximal de `a` (celui le plus bas de la branche la plus à droite de `a`).

On raisonne par récurrence sur la taille $n \geqslant 1$ de `a`.
Si $n = 1$ alors `a=Noeud(Vide,v,Vide)` et `(w,b)=(v,Vide)` et les propriétés sont bien satisfaites.
Supposons les propriétés satisfaites jusqu'à l'ordre $n-1$ avec $n \geqslant 2$ et soit `a=Noeud(g,v,d)` de taille n.
- Si `d=Vide` alors `(w,b)=(v,g)` et les propriétés sont bien satisfaites.
- Sinon `(w,b)=(vr,Noeud(g,v,ar))` avec `(vr,ar)=aux d`

1/ Par hypothèse de récurrence le calcul de `aux d` se termine avec un nombre d'appels récursifs égal à la longueur de la branche la plus à droite de `d` donc celui de `aux a` se termine bien et avec un nombre d'appels récursifs égal à la longueur de la branche la plus à droite de `a`.
2/ Par hypothèse de récurrence `Noeud(ar,vr,Vide)` est une réorganisaton de `d` en un ABR avec `vr` élément maximal de `d`. Donc `Noeud(b,w,Vide)=Noeud(Noeud(g,v,ar),vr,Vide)` est bien une réorganisation de `a`. C'est bien un ABR car `Noeud(g,v,ar)` est un ABR car `g` et `ar` le sont et car les étiquettes de `ar` sont parmi celles de `d` donc sont strictement supérieures à `w`. En outre `vr` est élément maximal de `d` donc de `a`.

Ce qui établit les 3 propriétés de manière générale. □

- Étudions désormais la fonction `eliminer` en prouvant que :
(1) elle se termine toujours;
(2) si `eliminer v a=r` alors `r` est un ABR tel que $C(r) = C(a) \setminus \{v\}$.

On raisonne par récurrence sur la taille n de `a`.
Si $n = 0$ i.e. si `a` est vide alors `r` est vide également et la propriété est bien vérifiée.
Supposons (1) et (2) vérifiées jusqu'à l'ordre $n-1$ avec $n \geqslant 1$ et soient `a` de taille n et un entier v.
Notons `a=Noeud(g,vp,d)` de sorte que $ABR(g) \wedge ABR(d)$, $x \leqslant vp$ si $x \in C(g)$ et $x > vp$ si $x \in C(d)$ (*)
Premier cas : $v < vp$
Notons `rg=eliminer v g` de sorte que `r=Noeud(rg,vp,d)`
Par hypothèse de récurrence le calcul de `eliminer v g` se termine donc celui de `eliminer v a` aussi et on a $ABR(rg)$ et $C(rg) = C(g) \setminus \{v\}$
Il découle alors immédiatement de (*) que $ABR(r)$ et que les étiquettes de r sont celles de `a` privé de `v` □
Deuxième cas : $v > vp$
Même démonstration, les rôles des fils `g` et `d` étant permutés. □
Troisième cas : $v = vp$
Par hypothèse de récurrence la calcul de `rg=eliminer v g` se termine.
- Si `rg` est vide alors `r=d` et la propriété est clairement satisfaite.
- Sinon `r=Noeud(gp,vm,d)` avec `(vm,gp)=aux rg`

D'après l'étude de la fonction `aux`, le calcul de `aux rg` se termine. Donc le calcul de `eliminer v a` se termine bien.
En outre `rg` vérifie (2) (par hypothèse de récurrence) et, d'après l'étude de la fonction `aux`, `Noeud(gp,vm,Vide)` est une réorganisation de rg en un ABR. Les éléments de `d` sont strictement supérieurs à `v` donc a fortiori à `vm` car `vm` est un élément de `g`. Il en résulte que `r=Noeud(gp,vm,d)` est bien un ABR et comme ses étiquettes sont celles de `a` privé de `v` la propriété (2) est établie. □

7) 8)
À chaque noeud visité la fonction `eliminer` est appelée sur le fils droit ou gauche. Le nombre d'appels récursifs est donc au plus égal à la profondeur de l'arbre.
Si la valeur `v` ne figure pas dans `a`, aucun appel à la fonction `aux`.
Si la valeur `v` figure une seule fois, la fonction `eliminer` appelle `aux` une seule fois ce qui déclenche (vu l'étude précédente) un nombre d'appel récursif de la fonction `aux` dominé par la profondeur de `a`
Si `v` figure m fois avec $m \geqslant 2$, la branche de gauche du noeud où il figure la première fois est filiforme avec dans l'ordre $m-1$ fois la valeur `v` puis éventuellement des valeurs strictement plus petites et décroissantes. Les appels à `aux` pour les noeuds d'étiquette `v` tombent donc alors directement sur le cas de base et n'induisent aucun appel récursif de la fonction `aux`.

Conclusion :

Dans tous les cas la complexité comptée en nombre d'appels récursifs des fonctions `eliminer` et `aux` est dominée par la profondeur de l'arbre.
1/ Si `a=Noeud(Vide,vp,g)` et si `v<vp` la complexité est minimale avec un seul appel récursif à la fonction `eliminer`
2/ Si `a=Noeud(g,v,d)` et que la branche la plus à gauche de `a` a pour longueur la profondeur de `a`, la complexité est un $\Theta(p)$ où p est la profondeur de l'arbre.
De manière plus précise la complexité maximale est de $p-1$ appels à `eliminer` et autant à `aux` si `g` ne contient pas la valeur `v` et si la branche la plus à droite de `g` a, comme sa branche la plus à gauche, pour longueur `p-1`. □

3/ Structure de données B-Arbres d'entiers

Remarque : Plutôt que de considérer l'ordre comme étant défini dans une variable globale, nous passerons cet ordre, noté `n`, en paramètre dans toutes les fonctions demandées.

Pour la commodité de lecture des fonctions on emploira les lettres suivantes (éventuellement indicées) :
- `a` pour un arbre (ou un noeud);
- `s` pour la séquence d'entiers d'une feuille;
- `f` pour la liste des paires d'un noeud (i.e. `freres`);
- `e` pour une étiquette;
- `p` pour une paire.

9)

```
let estComplet n a = match a with
  | Feuille s    -> ((list_length s) + 1) = 2 * n
  | Noeud (a1,f) -> ((list_length f) + 1) = 2 * n
;;
estComplet : int -> bArbre -> bool = <fun>
```

10) Cette fonction s'écrit facilement en récursivité croisée avec la fonction `taille_freres`.

```
let rec taille a = match a with
  | Feuille s   -> list_length s
  | Noeud (a1,f) -> taille a1 + taille_freres f
and taille_freres f = match f with
  | []           -> 0
  | (_,a)::suite -> 1 + taille a + taille_freres suite
;;
taille : bArbre -> int = <fun>
taille_freres : freres -> int = <fun>
```

11) De même récurvité croisée avec `recherche_freres`.

```
let rec recherche v a = match a with
  | Feuille [] -> false
  | Feuille (e::suite)  -> if (v < e) then false
                           else if (v = e) then true
                           else recherche v (Feuille suite)
  | Noeud (a,f) -> if recherche v a then true else recherche_freres v f
and recherche_freres v f = match f with
  | [] -> false
  | (e,a)::suite -> if (v < e) then false
                    else if (v = e) then true
                    else if recherche v a then true
                    else recherche_freres v suite
;;
recherche : int -> bArbre -> bool = <fun>
recherche_freres : int -> freres -> bool = <fun>
```

12)

```
let rec scissionBArbre a =  match a with
  | Feuille s -> let (s1,e,s2) = scissionSequence s in
                 (Feuille s1,e,Feuille s2)
  | Noeud (a,f) -> let (f1,p,f2) = scissionSequence f in
                   let a1 = Noeud (a,f1) and (e,a2) = p in
                     let a3 = Noeud (a2,f2) in
                       (a1,e,a3)
;;
scissionBArbre : bArbre -> bArbre * int * bArbre = <fun>
```

13) Le parcours commence à la racine qui est noeud complet et qui est donc scindé en l'arbre Noeud(a1,[(12;b1)]) avec a1=Noeud(a2,[(9;b2)]).
Comme $3 < 12$ le parcours se dirige sur la branche de gauche c'est à dire l'arbre a1 dont la racine est incomplète (une seule étiquette 9) donc pas de scission.
Puis $3 < 9$ et le parcours se dirige sur l'arbre a2=Noeud(Feuille([2;4;6]),[(7,b3)]).
Comme $3 < 7$ le parcours atteint la feuille Feuille([2;4;6]) qui est complète et est donc scindée en Noeud(Feuille([2]),[(4,Feuille([6]))]).
Puis enfin $3 < 4$ et 3 est inséré dans la feuille de gauche qui devient Feuille([2;3]).

14) 15)
Montrons par itération que l'algorithme se termine et préserve la structure de B-Arbre.
Notons $r_0 = a$ l'arbre initial et $p_0 = p$ sa profondeur et notons p_k la profondeur du noeud a_k sur lequel se dirige le parcours à un certain stade de l'algorithme et r_k l'état de l'arbre total à ce stade.
Supposons que r_k soit un B-Arbre.

- Si $p_k = 0$ alors a_k est la feuille cible.

1/ Si a_k est incomplète on insère v dans cette feuille qui devient a_{k+1}. L'algorithme est terminé et r_{k+1} obtenu à partir de r_k en remplaçant a_k par a_{k+1} est bien un B-Arbre. □
2/ Si a_k est complète, a_k est scindée en Noeud(f1,[(w,f2)]) où f1 et f2 sont deux feuilles incomplètes. L'arbre total devient r_{k+1} dont la structure de ABR est clairement préservée.
Puis le parcours se dirige sur f1 ou f2 suivant que v est inférieur ou supérieur à w et on retrouve dans le cas 1/ précédent d'une feuille incomplète. □

- Si $p_k \geqslant 1$ c'est à dire si a_k est un noeud interne :

1/ Si a_k est incomplet : on parcourt de gauche à droite la liste des étiquettes de a_k pour déterminer le fils noté a_{k+1} de a_k vers lequel se diriger (algorithme "bon-fils" décrit plus bas). Ainsi r_k n'est pas modifié donc la structure de ABR est bien conservée et p_k a diminué d'une unité.
2/ Si a_k est complet on le scinde en a'_k de profondeur $p_k + 1$. L'arbre total devient r_{k+1} qui conserve bien la structure de ABR puis le parcours se dirige sur le fils a_{k+1} droit ou gauche de la racine de a'_{k+1} de profondeur $(p_k + 1) - 1 = p_k$.
On se retouve alors dans la situation de 1/ et à l'étape suivante p_k diminue donc de une unité.

En conclusion tant que l'on ne tombe pas sur la feuille cible p_k diminue à chaque étape de une unité donc finit par atteindre 0 et l'algorithme se termine bien en renvoyant un B-arbre contenant la nouvelle valeur. □

Décrivons l'algorithme "bon-fils" sur un noeud incomplet `a=Noeud(a1,f)`
Notons que comme `a` n'est pas une feuille, la liste `f` est non vide. Notons `f=(e1,a2)::suite`
Si $v \leqslant e_1$ le bon fils est `a1`
Sinon Si `suite` est vide le bon fils est `a2`
Sinon on applique "bon-fils" à `Noeud(a2,suite)` □

16) Le rôle de "bon-fils" est tenu par la fonction `ajouter_freres v n f` dont les paramètres sont pour `f` une séquence non vide de paires (dont les arbres sont des ABR d'ordre `n`) et `v` un entier strictement plus grand que l'étiquette de la première paire de la séquence `f` et qui renvoie la séquence obtenue à partir de `f` en ajoutant l'entier `v` dans le bon arbre de la séquence et en préservant la structure de ABR.
Cette fonction est définie en récursivité croisée avec la fonction `ajouter v n a`.

```
let rec ajouter v n a = match a with
  | Feuille s when not (estComplet n a) -> Feuille (ajoutSequence v s)
  | Feuille s -> let (f1,w,f2) = scissionBArbre n a in
      ajouter v n (Noeud (f1,[(w,f2)]))
  | Noeud _ when (estComplet n a) -> let (a1,w,a2) = scissionBArbre n a in
      ajouter v n (Noeud (a1, [(w,a2)]))
  | Noeud (a1,f) -> let (e1,a2) = hd f in
      if (v <= e1) then Noeud(ajouter v n a1, f)
      else Noeud(a1,ajouter_freres v n f)
and ajouter_freres v n f = match f with
  | [(e,a)] -> [(e,ajouter v n a)]
  | (e1,a1)::(e2,a2)::suite ->
      if (v <= e2) then (e1,ajouter v n a1)::(e2,a2)::suite
      else (e1,a1)::(ajouter_freres v n ((e2,a2)::suite))
;;
ajouter : int -> int -> bArbre -> bArbre = <fun>
ajouter_freres : int -> int -> freres -> freres = <fun>
```

FIN