Table des matières

MPSI : Troisième devoir d'option informatique

Durée : quatre heures. Le sujet comporte deux problèmes indépendants : le premier sur les arbres et le second sur les piles. La fin du second problème (version récursive et amélioration) est facultative.

1 Arbres lexicographiques

solution : **Erreurs fréquentes :** Ne pas comprendre le rôle du mot vide, et d'un nœud terminal à la racine. Dans cette situation, vous êtes tentés de mettre les cas d'arrêts aux mots de une lettre, ce qui est lourd.

On représentera les mots sous Caml non pas par le type `string` mais une liste de caractères.

```
type mot = char list;;
```

Exemple : L'expression `[ 'f'; 'a'; 'c'; 'e' ]` est associée au mot « face », de longueur 4.
N.B. En Caml, un caractère est entouré d'apostrophes, (touche 4 d'un clavier azerty), tandis qu'une chaîne de caractères est entourées de guillemets anglais doubles (touche 3).

Les arbres lexicographiques sont des arbres utilisés pour représenter des ensembles de mots. Les arêtes sont étiquetés par un caractère. La séquence des caractères qui étiquettent les arêtes le long d'un chemin de la racine de l'arbre jusqu'à un nœud forme donc un mot.

Certains nœuds sont appelés nœuds « terminaux ». Les mots représentés par l'arbre sont les mots obtenus en suivant un chemin de la racine jusqu'à un nœud terminal.

Attention : un nœud terminal n'est pas forcément une feuille.

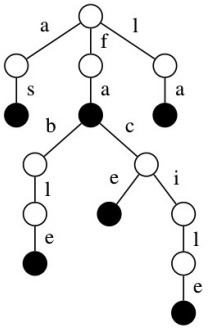

FIGURE 1 – Un exemple d'arbre lexicographique représentant l'ensemble {"as", "fa", "fable", "facile", "la"}. Les nœuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche.

Un arbre lexicographique est représenté par le type `arbre_lex` et le type auxiliaire `fils` ci-dessous :

```
type arbre_lex = Noeud of bool * fils
and fils = (char * arbre_lex) list;;
```

Dans l'appel `Noeud(terminal, fils)`, `terminal` est un drapeau booléen qui indique si le nœud est terminal ou pas. Et `fils` contient la liste des fils ainsi que les étiquettes des arêtes qui y mènent. C'est une liste de couples de la forme `(c, f)` tel que `c` est l'étiquette de l'arête menant au fils `f`.

En outre, un arbre lexicographique sera dit *valide* si pour chaque nœud, les fils sont rangés dans l'ordre alphabétique des étiquettes des arêtes qui y mènent, et si toutes les feuilles sont des nœuds terminaux.

Sauf mention du contraire, on supposera que tous les arbres lexicographiques utilisés sont valides, et dans les question demandant de créer un arbre, on veillera à créer un arbre valide.

Remarque : Ainsi les mots dans un arbre lexicographiques sont-ils rangé dans l'ordre alphabétique. Un tel arbre est donc semble à un arbre binaire de recherche, à la différence près qu'il n'est pas binaire, puisqu'un nœud peut avoir autant de fils qu'il y a de lettres dans l'alphabet choisi.

1. Écrire une fonction `mot_of_string` de type `string -> mot` prenant en entrée une chaîne de caractère, de type `string` et la convertissant en une liste de lettres. On pourra écrire une fonction récursive auxiliaire prenant un argument supplémentaire : la position actuelle dans la chaîne de caractères.
   solution :

```
let rec mot_of_stringAux i m=
  (* Renvoie la liste de caractères [m.[i]; ...; m.[n-1]], où n = String.length m *)
  if i >= String.length m then []
  else m.[i] :: mot_of_stringAux (i+1) m;;

let mot_of_string = mot_of_stringAux 0;;
```

2. L'expression suivante définit un arbre `exemple` :

```
  let feuille= Noeud(true,[]);;
  let exemple=
  Noeud(false,[
      'f',Noeud(false,[
          'a', Noeud(true,[
               ('c', feuille)
                          ]
                     )
      ;
          ('i', feuille)
          ])
  ]);;
```

   Dessiner l'arbre lexicographique correspondant, et donner l'ensemble de mots qu'il représente.

3. Dans quelle situation un nœud terminal qui n'est pas une feuille est-il utile ?
   solution : Lorsqu'un mot est préfixe d'un autre.

4. Dans quelle situation la racine est-elle un nœud terminal ?
   solution : Lorsque le mot vide est dans l'ensemble de mots à représenter. **N.B.** Cette situation sera utile lors des appels récursifs des fonctions à venir.

5. Écrire en CaML une fonction `arbre_of_mot` de type `mot -> arbre_lex` telle que l'appel `(arbre_of_mot m)` sur un mot `m` renvoie un arbre lexicographique contenant uniquement le mot `m`.
   solution :

```
let rec arbre_of_mot = function
  |[]-> feuille
  |t::q -> Noeud(false, [(t, arbre_of_mot q)])
;;
```

6. Écrire en CaML une fonction `nb_mots` de type `arbre_lex -> int` telle que l'appel `(nb_mots a)` sur un arbre lexicographique `a` renvoie le nombre de mots contenus dans l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
   solution : Il s'agit tout simplement de compter le nombre de nœuds terminaux.

```
let rec nb_mots (Noeud(term, fils)) = (* Comme il 'ny a 'quun seul constructeur pour le
    ↪ type arbre, je peux utiliser le raccourci ci-contre. *)
    if term then 1 + nb_mots_foret fils
    else nb_mots_foret fils

and nb_mots_foret = function
    |[]   -> 0
    |(_,a)::q  -> nb_mots a + nb_mots_foret q
;;
```

7. Écrire également une fonction calculant le nombre de feuilles d'un arbre lexicographique.
solution :

```
''
let rec nb_feuilles  a =
  match a with
    | Noeud(_,[])  -> 1
    | Noeud(_,fils)-> nb_feuilles_foret fils

and nb_feuilles_foret lf =
    match lf with
    |[]   -> 0
    |(_,a)::q  -> nb_feuilles a + nb_feuilles_foret q
;;
```

8. Écrire une fonction `prefixed` prenant un caractère x et une liste de mots l et remvoyant la liste obtenue en ajoutant x devant chaque mot de l. Bonus à qui l'écrit en une ligne.
solution :

```
''
let prefixed x (l:mot list)=
    (* Renvoie la liste contenant les mots de l prefixée 'dun x. *)
    List.map (fun m-> x::m) l;;
```

9. Écrire en CaML une fonction `mots_of_arbre` de type `arbre_lex -> mot list` telle que l'appel `(mots_of_arbre a)` sur un arbre lexicographique `a` renvoie la liste des mots contenus dans `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
'''
let rec mots_of_arbre a=
    (* Renvoie la liste des mots contenus dans a. *)
    match a with
    |Noeud(true, fils)-> [] :: mots_of_arbre_foret fils (* le mot vide est contenu dans
        ↪ a*)
    |Noeud(false, fils)-> mots_of_arbre_foret fils

and mots_of_arbre_foret f=
  match f with
  |[]-> []
  |(lettre, a)::q -> (prefixed lettre (mots_of_arbre a))
                    @ mots_of_arbre_foret q
;;
```

10. Écrire en CaML une fonction `appartient` de type `mot -> arbre_lex -> bool` telle que l'appel `(appartient m a)` sur un mot `m` et un arbre lexicographique valide `a` renvoie la valeur `true` si et seulement si l'arbre `a` contient le mot `m`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives. La complexité de la recherche d'un mot m doit être en $O(|m|)$. On justifiera cette complexité.
solution : Comme remarqué question **??**, ne pas oublié le cas du mot vide. Dans ce cas, renvoyer le booléen qui est à la racine.

```
''
let rec appartient (m:mot) a =
    (* Indique si le mot m est contenu dans l'arbre a. *)
    match m, a with
    |[], Noeud(term,_) -> term (* Le mot vide est reconnu ssi la racine est terminale *)
    |t::q, Noeud(_,fils)-> appartient_foret t q fils

and appartient_foret lettre mot f =
   match f with
   |[] -> false
   |(x,a)::q when x=lettre -> appartient mot a
   |(x,_)::q when x<lettre -> appartient_foret lettre mot q
   |_ -> false
;;
```

11. Écrire une fonction `ajout` de type `mot -> arbreLex -> arbreLex` prenant en entrée un mot `m` et un arbre lexicographique `a` et renvoyant l'arbre obtenu en rajoutant `m` dans `a`.
solution :

```
'''
let rec ajout m a=
    match m, a with
    |[], Noeud(_, fils)-> Noeud(true,fils)
    |t::q, Noeud(term, fils)  -> Noeud(term, ajout_foret t q fils)

and ajout_foret lettre mot f=
   match f with
   |[] -> [(lettre, arbre_of_mot mot)]
   |(x,a)::q when lettre <x  -> (lettre, arbre_of_mot mot)::f
   |(x,a)::q when lettre=x   -> (x, ajout mot a)::q
   |(x,a)::q                 -> (x,a):: ajout_foret lettre mot q
;;
```

12. En déduire une fonction `arbre_of_list` prenant une liste de mots et renvoyant un arbre lexicographique contenant les mêmes mots.
solution : On peut le faire en une ligne en utilisant un `List.fold` avec l'opération `ajout`.

```
'''
let arbre_of_list (l:mot list)=
    List.fold_right ajout l (Noeud(false,[]))
;;
```

13. Le fichier `mots_francais.txt` contient tous les mots du français, un par ligne. Voici les commandes de base pour manipuler un fichier :
    - Ouvrir un fichier : `let entree = open_in "adresse du fichier"in ...`
      L'identifiant `entree` contient alors un objet de type « `in_channel` ».
    - Lire une ligne : `input_line` de type `in_channel -> string`. Cette fonction lève l'exception `End_of_file` lorsqu'on est au bout du fichier.
      On pourra utiliser la fonction suivante :

```
let lit_une_ligne entree=
  (* Renvoie la prochaine ligne du canal d'entrée fourni privé de son dernier
      ↪ caractère (le retour chariot), ou "" si on est au bout du fichier. *)
  try
    let ligne = input_line entree in
    let n = String.length ligne in String.sub ligne 0 (n-1)
  with
  |End_of_file -> ""
;;
```

      Pour des raisons de performance, puisque le fichier à lire est gros, il faudra écrire une fonction auxiliaire avec accumulateur.

    Écrire une fonction pour charger tous les mots du français dans un arbre lexicographique.
    solution :

```
'''
let arbre_of_txt chemin=
  let entree=open_in chemin in

  let rec lit_fichier accu=
    let ligne = lit_une_ligne entree in
    if ligne = "" then accu
    else lit_fichier (ajout (mot_of_string ligne) accu)
  in
  lit_fichier (Noeud(false, []));;
```

14. En déduire une fonction `est_francais` permettant de tester si un mot est français.
solution :

```
1 '''
199 let est_francais m = appartient (mot_of_string m) dico_francais;;
200 (*-fin -*)
201 est_francais "Noël";;
202  est_francais "élève";;
203 est_francais "Banach";;
204 est_francais "ban";;
```

Dans un chapitre ultérieur, nous verrons comment calculer la « distance » entre deux mots, c'est-à-dire le nombre de fautes de frappe pour passer de l'un à l'autre. En combinant cela avec ce que nous venons de faire, on obtient un correcteur orthographique.

2 Piles : résolution d'un sudoku

2.1 Généralités

Un sudoku est une matrice de format 9×9. Chaque case peut être remplie par :

- Un entier de l'intervalle $[\![1, 9]\!]$;
- Un zéro, auquel cas nous dirons que la case est vide.

```
1 type sudoku = int array array;;
```

Un sudoku s est valide s'il vérifie les trois conditions suivantes :

- Pour tout $n \in [\![1, 9]\!]$, n ne peut apparaître deux fois dans une même ligne ;
- Pour tout $n \in [\![1, 9]\!]$, n ne peut apparaître deux fois dans une même colonne ;
- On divise la matrice s en 9 blocs de format 3×3 comme sur la figure :

7	6	3	8	9	2	5	4	1
9	2	1	7	5	4	8	6	3
5	4	8	1	3	6	9	7	2
6	5	4	2	1	7	3	8	9
2	8	9	4	6	3	1	5	7
3	1	7	9	8	5	4	2	6
4	9	2	3	7	8	6	1	5
8	3	5	6	2	1	7	9	4
1	7	6	5	4	9	2	3	8

On demande alors que pour tout $n \in [\![1, 9]\!]$, n n'apparaisse pas deux fois dans un même bloc.

Si de plus aucune case de s n'est vide, on dira que s est résolu (l'exemple ci-dessus montrait un sudoku résolu).

Notre but est d'écrire un programme prenant un sudoku valide et le remplissant si possible afin d'obtenir un sudoku résolu.

Nous allons suivre la méthode naïve, employée par la plupart des joueurs : on choisit une case, on y inscrit un chiffre qui ne contredise aucune des trois règles ci-dessus puis on passe à une case suivante. Et si on arrive à une impasse (aucune possibilité pour une certaine case) il faut effacer les chiffres inscrits et recommencer avec d'autres.

La première étape est, étant donnés un sudoku s et une case (i, j) d'icelui, de calculer tous les chiffres pouvant être inscrits dans cette case tout en satisfaisant les trois règles du jeu. Pour ce, on va créer un tableau `possible` de 10 cases, initialement toutes vraies, puis pour tout $n \in [\![1, 9]\!]$, on mettra faux dans `possible.(n)` si on ne peut par mettre n dans la case $s.(i).(j)$.

On pourra supposer que la case (i, j) de s est vide.

1. Écrire une procédure `regleLigne` de type `sudoku -> int -> int array -> unit` prenant en entrée le sudoku s, l'indice de ligne i, et le tableau `possible`, et passant `possible.(n)` à `false` pour tout n qu'on ne peut inscrire en case (i, j) de s sans contrevenir à la règle sur les lignes.
   solution :

```
let regle_ligne s i possible=
    for j = 0 to 8 do
      possible.(s.(i).(j)) <- false
    done
;;
```

2. La procédure `Array.iter` de type `('a -> unit)-> 'a array -> unit` prend en entrée une procédure f et un tableau t et applique f à chaque élément de t.
Programmer à l'aide de `Array.iter` une version de `regleLigne` sans boucle explicite.
solution :

```
let regle_ligne s i possible=
  Array.iter (fun n -> possible.(n) <- false)
             s.(i)
;;
```

3. Les neuf blocs de format 3×3 utilisés par la troisième règle seront repérés par un couple $(ib, jb) \in [\![0, 3[\![^2$ de la manière naturelle. Par exemple la case $(2, 1)$ de s est dans le bloc $(0, 0)$. La case $(3, 7)$ est dans le bloc $(1, 2)$.
Quelles opérations simples permettent de trouver les indices (ib, jb) du bloc contenant la case (i, j) de s ?
solution : On utilise une division euclidienne par 3 : `ib = i/3`et `jb = j / 3`.
4. Écrire une procédure `regleBlocs` analogue aux précédente pour prendre en compte la troisième règle.
solution :

```
let regle_bloc s (i0, j0) possible=
  let ib, jb = i0/3, j0/3 in
  for i = 3*ib to 3*ib+2 do
    for j = 3*jb to 3*jb+2 do
      possible.(s.(i).(j)) <- false
    done
  done
;;
```

5. Écrire enfin une fonction `valeursPossibles` de type `sudoku -> (int*int)-> bool array` prenant en entrée un sudoku s et les coordonnées (i, j) d'ure de ses cases, et renvoyant le tableau `possible` indiquant les valeurs pouvant être inscrites dans la case `s.(i).(j)` sans contrevenir aux trois règles.
solution :

```
let valeurs_possibles s (i,j) =
  let possible = Array.make 10 true in
  regle_ligne s i possible;
  regle_colonne s j possible;
  regle_bloc s (i,j) possible;
  possible;;
```

6. Combien de lectures de cases du sudoku sont-elles effectuées lors d'un appel à `valeursPossibles` ?
solution : Chacune des procédures `regle_ligne`, `regle_colonne` et `regle_bloc` nécessite de lire neuf cases du sudoku. Nous lisons donc au final vingt-sept cases.

2.2 Version impérative

On commence par une version impérative. Nous utiliserons trois piles mutables `aRemplir`, `remplies`, et `aEssayer`.

- `aRemplir` contiendra les cases à remplir, et `remplies` les cases déjà remplies. Ces piles contiendront donc des couples d'entiers.
- `aEssayer` contiendra les prochains essais à faire. Ce sera une liste de triplets : un triplet (i, j, n) signifiera que l'essai à effectuer est d'inscrire n dans la case (i, j).

Nous utiliserons les piles du module `Stack` de Caml. En plus des fonctions `create, pop, push`, et `is_empty` déjà utilisées en cours, on pourra utiliser `top` qui renvoie l'élément au sommet d'une pile mais sans le dépiler (c'est donc une fonction pure et non une « fonction-procédure » comme `pop`).

1. Écrire une fonction `init_aRemplir` de type `sudoku -> (int*int)Stack.t` qui prend un sudoku s et qui crée et renvoie la pile `aRemplir` en y mettant toutes les cases vides de s.
solution :

```
let init_a_remplir s=
  let aRemplir = Stack.create () in
  for i=0 to 8 do
    for j=0 to 8 do
      if s.(i).(j) = 0 then Stack.push (i,j) aRemplir
    done
  done;
  aRemplir
;;
```

2. Écrire une procédure `empilePossibles` de type `sudoku -> (int*int)-> (int * int * int)Stack.t -> unit` qui prendra en entrée le sudoku s, les coordonnées (i, j) d'une case, et la pile `aEssayer`, et qui empilera dans `aEssayer` tous les essais possibles en case (i, j).
solution :

```
let empile_possible s (i,j) p =
  let possible=valeurs_possibles s (i,j) in
  for n = 1 to 9 do
    if possible.(n) then
      Stack.push (i,j,n) p
  done
;;
```

3. Le point clef est la phase de retour en arrière, lorsqu'on efface les essais infructueux. Il faudra dépiler les éléments de `remplies`, les remettre dans `aRemplir` en effaçant au passage la case correspondante dans s, et ce jusqu'à ce que la prochaine case à remplir corresponde au prochain essai contenu dans la pile `aEssayer`.
Écrire une fonction `retour_arriere` correspondante. Elle sera de type `sudoku -> (int * int)Stack.t -> (int * int)Stack.t -> int * int -> unit` : elle prendra en arguments le sudoku s, les piles `remplies` et `aRemplir`, ainsi que les coordonnées (i, j) de la case intervenant dans le prochain essai contenu dans `aEssayer`.
solution :

```
let retour_arriere s remplies aRemplir case =
  (* case est la case au sommet de aEssayer, càd la prochaine case pour laquelle on a des possibilités. *)
  (* A l'issue de cette procédure, case est la case au sommet de aRemplir *)
  while case <> Stack.top aRemplir do
    let (i,j) = Stack.pop remplies in
    s.(i).(j) <- 0;
    Stack.push (i,j) aRemplir
  done;;
```

4. Passer enfin à la fonction finale ! À chaque étape de la boucle, il faudra :
   - Calculer et empiler tous les essais possibles pour la prochaine case à remplir (c'est-à-dire le sommet de `aRemplir`) ;
   - Effectuer le retour arrière, dans le cas où aucune possibilité n'a été trouvée à l'étape précédente ;
   - Effectuer le prochain essai, ce qui impose de mettre à jour les piles ainsi que s.

   On supposera le sudoku résoluble : on ne se préoccupera donc pas de lever une exception dans le cas où la résolution n'aboutit pas.
solution :

```
let resolution s=
  (* Remplit s par effet de bord. *)

  (* Initialisation des variables *)
  let aRemplir = init_a_remplir s in (* pile des cases à remplir *)
  let remplies = Stack.create () in
  let aEssayer  = Stack.create () in (*  Syntaxe : (x,y,n), où (x,y) = coord de la case, et n=nombre à écrire. *)
```

```
163
164
165   while not (Stack.is_empty aRemplir) do
166
167     (* Calcul des possibilités pour la case suivante. *)
168     empile_possible s (Stack.top aRemplir) aEssayer;
169
170     let (i,j,n) = Stack.pop aEssayer in
171
172     (* Retour arrière jusqu'à ce que prochaineCase corresponde au prochain essai à faire.
        ↪ *)
173     (* Note : On peut le lancer dans tous les cas : il produira un effet uniquement si
        ↪ aucune possibilité n'avait été trouvé lors de la première phase l'itération. *)
174     retour_arriere s remplies aRemplir (i,j);
175
176     (* Faire le prochain essai. *)
177     s.(i).(j) <- n;
178     let _= Stack.pop aRemplir in
179     Stack.push (i,j) remplies;
180   done
181 ;;
```

2.3 Version Récursive

On passe maintenant à une version récursive, toujours basée sur le même principe de faire des essais et d'effacer les essais n'ayant pas permis de terminer la résolution.

On s'interdira l'usage de boucles, et la création de variable, et plus généralement de tout objet mutable. On pourra quand même utiliser les programmes créés dans la première partie, ainsi que le sudoku s lui-même.

Je ne vous donne aucune question intermédiaire mais un squelette de code à compléter :

```
let liste_possibles s (i,j) =
  (* Renvoie la liste des valeurs pouvant être inscrites dans s.(i).(j) sans contrevenir aux
     ↪ règles.*)

  let possible = valeurs_possibles s (i,j) in

  let rec aux n =
     (* n est la prochaine valeur à essayer. *)

  ...
;;
```

```
let rec prochaine_case (i, j) s =
  (* Renvoie (-1, -1) si toutes les cases de s sont remplies, et renvoie la prochaine case
     ↪ vide de s dans le cas contraire.
     L'argument supplémentaire (i, j) indique la prochaine case à tester. *)
  ...
;;
```

Le cœur du programme est formé de deux fonctions mutuellement récursives : la fonction principale, et une fonction chargée de parcourir toutes les possibilités en une case donnée. (Même structure que pour le calcul des permutations fait en TP : il s'agit dans le fond de parcourir l'arbre des essais possibles...)

Ces fonctions renvoient en outre un booléen pour indiquer si la résolution a abouti, ce qui permettra de savoir quand il faut annuler l'essai effectué.

```
let rec resolution_rec s =
  (* Renvoie un booléen qui indique si le sudoku s est résoluble.
     En outre, si tel est le cas, remplit s en effet de bord. *)
  match prochaine_case 0 0 s with
  |(-1, -1) -> ...
  |(i,j)    -> fait_les_essais s (i,j) (liste_possibles s (i,j) 9)

```

```
8
9  and fait_les_essais s (i,j) l=
10 (* Essaie de mettre, chacun à son tour, tous les éléments de l à la case indiquée.
11    Renvoie true si une des possibilité permet de terminer la résolution et false sinon.*)
12   match l with
13   | []   -> ...
14   | t::q ->
15      begin
16        s.(i).(j) <- t;
17        if resolution_rec_aux s then ...
18        else ...
19      end
20 ;;
```

Commentaire : Ce type de méthode consistant à essayer toutes les possibilités, en revenant en arrière dès qu'on détecte une impossibilité s'appelle en anglais une méthode par « backtracking » (« backtracking » signifie « retour en arrière »). Une telle méthode peut être implémentée par une pile, ou par une fonction récursive. De manière générale, une fonction récursive peut toujours être remplacée par une fonction non récursive utilisant une pile.

2.4 Amélioration

Une amélioration consiste à choisir comme prochaine case à remplir celle où il y a le moins de choix. C'est ce que fait en général un joueur humain.

Expliquer comment on pourrait modifier les programmes précédents pour effectuer cette modification. Quelle version s'y prête le mieux ?

solution : Dans la version récursive, il suffit de modifier la fonction `prochaine_case` pour qu'elle renvoie les coordonnées de la case ayant le moins de possibilités.

D'ailleurs, si on veut optimiser encore, il faudrait garder en mémoire à chaque instant la liste des possibilités dans chaque case, pour éviter de devoir les recalculer sans arrêt, surtout au moment de choisir la case suivante.

Pour la version impérative, ce sera plus compliqué. On peut tout de même sans trop d'efforts au moment de l'initialisation de la pile prendre soin de la trier par ordre décroissant de nombre de possibilités.

2.5 Troll

Qui est le plus fort : un programme impératif ou récursif ?

solution : Les travaux de Church et Turing prouvent que toute fonction pouvant être calculée par un programme impératif peut aussi l'être par un programme récursif.