CCP 2002 Option Informatique : Algorithmique et Programmation en CaML.

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

On commence par définir un type boolean pour simuler les valeurs binaires 0 et 1 (alias `false` et `true`) puis les types synomymes définis dans l'énoncé :

```
type boolean = O | I;;

type suite == char list
and  code  == boolean list
and  cle   == (char * code) list;;
```

Questions 1, 2 et 3.
Codage d'une suite de caractères à l'aide d'une clé de codage.

```
let rec coder_char (a:char) (clef:cle) = match clef with
  | [] -> failwith "caractère non présent dans la clé !"
  | (x,xx) :: suite -> if x = a then xx else coder_char a suite
;;
coder_char : char -> (char * code) list -> code = <fun>

let rec coder (texte:suite) (clef:cle) = match texte with
  | [] -> []
  | a :: suite -> (coder_char a clef) @ (coder suite clef)
;;
coder : char list -> cle -> boolean list = <fun>
```

La fonction `coder_char a clef` renvoie le code du caractère a selon la clé `clef`. Elle parcourt pour cela la liste `clef` jusqu'à trouver l'élément `(a,aa)` et renvoie aa (liste de booléens). Sa complexité est $O(|clef|)$.

La fonction `coder texte clef` parcourt la liste `texte` en concaténant le code de chacun des caractères. Elle effectue exactement $|texte|$ appels récursifs à elle-même, chacun étant suivi d'un appel à la fonction récursive de codage du caractère rencontré (de complexité $O(|clef|)$) puis de la concaténation. Si l'on ne prend pas en compte la concaténation, la complexité est donc $O(|texte| \times |clef|)$.
En fait la complexité réelle (en prenant en compte la concaténation) est :
$O(|texte| \times (|clef| + |code|))$ où $|code|$ est la longueur moyenne d'un code.

Question 4.

Soit `clef` une clé de codage associée à un arbre `arbre` complet et soit $b = b_1 \dots b_r \in \mathfrak{I}$.
Soit $i \in [1, r]$. Alors b_i correspond à une arête issue d'un noeud interne de `arbre`. Comme l'arbre est complet, il existe donc l'arête $b'i = \neg b_i$ issue du même noeud interne. Cette arête parmet d'atteindre une (ou plusieurs) feuille. En d'autres termes il existe $b'_{i+1}, \dots, b'_s$ tels que $b' = b_1 \dots b_{i-1} b'_i b'_{i+1} \dots b'_s \in \mathfrak{I}$. Donc `clef` est optimale. □

Question 5.

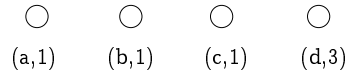
Par définition même d'une clé de codage séparable, une clé est séparable si et seulement si les noeuds internes de l'arbre associé ne contiennent aucun caractère. □

Question 6.

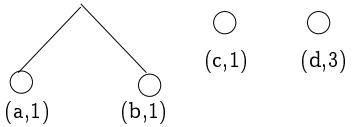
```
let rec nombre a = match a with
  | Vide -> 0
  | Feuille(_,n) -> n
  | Noeud(a1,a2) -> (nombre a1) + (nombre a2)
;;
nombre : arbre -> int = <fun>
```

Question 7.
Exemple de construction d'un arbre de Huffman associé à une suite de caractères.

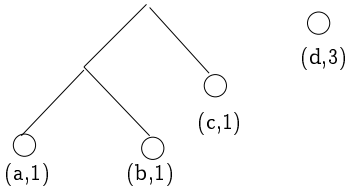
On remarquera que l'arbre est unique à une permutation près entre les feuilles correspondant à des caractères de même occurence.



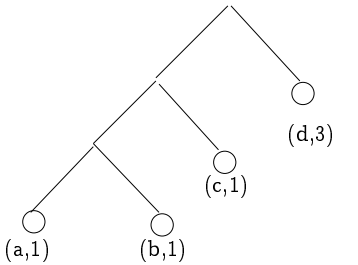
Étape 0.



Étape 1.



Étape 2.



Étape 3 (finale).

Remarque.

La terminologie *arbre de Huffman* de l'énoncé peut prêter à confusion.
Dans la suite on désignera par arbre de codage un arbre associé à une clé de codage optimale et séparable. Il s'agit donc d'un arbre binaire complet dont seules les feuilles contiennent un caractère (avec éventuellement un nombre d'occurences).
On désignera par arbre de Huffman un arbre de codage obtenu à l'aide de l'algorithme de Huffman.

Question 8.

On a $C(H) = \sum_{k=1}^{p} C(\nu_k)O(\nu_k) = C(\nu_i)O(\nu_i) + C(\nu_j)O(\nu_j) + \sum_{k\neq i,j} O(\nu_k)C(\nu_k)$
où $C(\nu_k)$ est le coût donc la profondeur du caractère ν_k.
Permuter les feuilles ν_i et ν_j revient à échanger leur coût donc :
$C(H') = C(\nu_j)O(\nu_i) + C(\nu_i)O(\nu_j) + \sum_{k\neq i,j} O(\nu_k)C(\nu_k)$
Donc $C(H) - C(H') = \Big(C(\nu_i) - C(\nu_j)\Big).\Big(O(\nu_i) - O(\nu_j)\Big)$. □

Question 9.

Soit H un arbre de codage minimal pour une suite de caractères.
Supposons qu'il existe une feuille ν_j de profondeur non maximale contenant un caractère d'occurence strictement plus petite que tous ceux contenus dans les feuilles de profondeur maximale. Soit ν_i une feuille quelconque de profondeur maximale.
Alors $C(\nu_i) - C(\nu_j) > 0$ et $O(\nu_i) - O(\nu_j) > 0$ donc $C(H) > C(H')$ d'après la question précédente en notant H' l'arbre obtenu à partir de H en échangeant les feuilles ν_i et ν_j.
Contradiction avec le fait que H soit un arbre de codage minimal. □

Question 10.

Puisque ν_i et ν_j ont le même père dans l'arbre de codage H, ces deux feuilles ont la même profondeur $p = C(\nu_i) = C(\nu_j)$.
Ainsi $C(H) = p\big(O(\nu_i) + O(\nu_j)\big) + K = pO(n) + K$ et $C(H') = (p-1)O(n) + K$ en notant K le coût total des caractères ν_k avec $k \neq i, j$.
Supposons que H' ne soit pas minimal. Alors il existe un arbre H'' de coût moindre pour la même suite de caractères que H'. Ce coût est $qO(n) + L < (p-1)O(n) + K$ où q est la profondeur de la feuille n et L le coût total des ν_k avec $k \neq i, j$.
Soit alors $\widetilde{H}$ l'arbre de codage de la même suite de caractères que H obtenu à partir de H'' en remplaçant la feuille n par deux fils débouchant sur les feuilles ν_i et ν_j. Le coût de $\widetilde{H}$ est alors $(q+1)\big(O(\nu_i) + O(\nu_j)\big) + L = (q+1)O(n) + L < pO(n) + K = C(H)$.
Contradiction avec le fait que H soit minimal. □

Question 11.

Commençons par remarquer qu'étant donné un texte (i.e. une suite de caractères) à coder il y a pseudo-unicité de l'arbre de codage minimal (unicité à une permutation près entre les feuilles correspondant à des caractères de même occurence dans le texte).
De même il y a pseudo-unicité de l'arbre de codage construit par l'algorithme de Huffmann.
Ainsi pour prouver que tout arbre de codage construit par l'algorithme de Huffman est minimal, il suffit de prouver que tout arbre de codage minimal peut être construit par l'algorithme de Huffman. On prouvera ainsi en fait que, pour un texte donné, l'ensemble des arbres de codage minimal est exactement égal à l'ensemble des arbres de codage construits par l'algorithme de Huffman.

On va établir ce résultat par récurrence sur le nombre p de feuilles d'un arbre de codage minimal (i.e. sur le nombre de caractères distincts de la suite à coder).
Tout arbre de codage minimal ayant 1 ou 2 feuilles est clairement construisible par l'algorithme de Huffman.
Supposons que tout arbre minimal ayant $p-1$ feuilles ($p \geqslant 3$) soit construisible par l'algorithme de Huffman.
Soit alors H un arbre de codage minimal ayant p feuilles et H' comme dans la question précédente. Alors puisque H' est minimal (question précédente) et possède $p-1$ feuilles, il est construisible par l'algorithme de Huffman ce qui revient exactement à dire que H est construit par l'algorithme de Huffman (par fonctionnement même de l'algorithme).
Ce qui établit le résultat. □

Question 12.

La fonction suivante renvoie `true` si le nombre total d'occurences de l'arbre de codage `a1` est inférieur ou égal à celui de `a2`.

```
let comparer a1 a2 = (nombre a1) <= (nombre a2);;
comparer : arbre -> arbre -> bool = <fun>
```

Questions 13 et 14.
Tri par insertion d'une suite d'arbres de codage.

Première solution.

Les deux fonctions suivantes sont très simples mais leur récursivité n'est pas terminale.

La fonction suivante insère l'arbre de codage `a` dans la liste triée d'arbres `l` (supposée ne pas contenir l'arbre `a`) en respectant l'ordre.

```
let rec insérer a l = match l with
  | []                    -> [a]
  | b :: suite when b = a -> failwith "déjà présent dans la liste !"
  | b :: suite            -> if comparer a b then a :: b :: suite
                             else b :: (insérer a suite)
;;
insérer : arbre -> arbre list -> arbre list = <fun>
```

La fonction suivante trie par insertion une liste d'arbres de codage.

```
let rec trier l = match l with
  | []         -> []
  | a :: suite -> insérer a (trier suite)
;;
trier : arbre list -> arbre list = <fun>
```

Seconde solution.

Les fonctions suivantes sont moins simples mais elles sont récursives terminales.
La fonction d'insertion fait appel à la fonction interne `rev` (qui renvoie la liste miroir d'une liste) et cette fonction est elle-même récursive terminale.

```
let rec inserer_aux a result_partiel reste = match reste with
  | []         -> result_partiel
  | b :: suite ->
        if comparer a b then result_partiel @ (b :: suite)
        else inserer_aux a (rev (a :: b :: tl (rev result_partiel))) suite
;;
inserer_aux : arbre -> arbre list -> arbre list -> arbre list = <fun>

let inserer a = inserer_aux a [a];;
inserer : arbre -> arbre list -> arbre list = <fun>
let rec trier_aux result_partiel reste = match reste with
  | []         -> result_partiel
  | a :: suite -> trier_aux (inserer a result_partiel) suite;;
trier_aux : arbre list -> arbre list -> arbre list = <fun>

let trier = trier_aux [];;
trier : arbre list -> arbre list = <fun>
```

Questions 15 et 16.
Construction de la liste des feuilles "occurencées" associée à une suite de caractères.

Première solution.

Fonctions simples récursives non terminales.

La fonction suivante ajoute le caractère v à la liste de feuilles l.

```
let rec ajouter v l = match l with
  | []                     -> [Feuille(v,1)]
  | Feuille (w,p) :: suite -> if w = v then Feuille (w,p+1) :: suite
                              else Feuille (w,p) :: ajouter v suite
  | _                      -> failwith "l est une liste de feuilles !"
;;
ajouter : char -> arbre list -> arbre list = <fun>
```

La fonction suivante renvoie une liste de feuilles correspondant à la suite de caractères s.

```
let rec compter s = match s with
  | []         -> []
  | v :: suite -> ajouter v (compter suite)
;;
compter : char list -> arbre list = <fun>
```


Seconde solution.

Fonctions récursives terminales.

```
let rec ajouter_aux v result_partiel reste = match reste with
  | []                     -> result_partiel
  | Feuille (w,p) :: suite ->
      if w = v then tl (rev result_partiel) @ (Feuille (w,p+1) :: suite)
      else ajouter_aux v (Feuille (w,p) :: result_partiel) suite
  | _                      -> failwith "l est une liste de feuilles !"
;;
ajouter_aux : char -> arbre list -> arbre list -> arbre list = <fun>

let ajouter v = ajouter_aux v [Feuille (v,1)];;
ajouter : char -> arbre list -> arbre list = <fun>
let rec compter_aux result_partiel reste = match reste with
  | [] -> result_partiel
  | v :: suite -> compter_aux (ajouter v result_partiel) suite
;;
compter_aux : arbre list -> char list -> arbre list = <fun>

let compter = compter_aux [];;
compter : char list -> arbre list = <fun>
```

Questions 17, 18 et 19.
Construction de l'arbre de Huffman associé à une suite de caractères.

La fonction suivante fusionne par l'algorithme de Huffman une liste non vide triée d'arbres de codage.
Lorsqu'on l'applique à une liste triée de feuilles (correspondant à des caractères distincts) on obtient donc un arbre de Huffman (donc minimal) pour la liste de caractères correspondant à la liste de feuilles.

La terminaison est assurée par le fait que la taille de la liste non vide diminue d'une unité à chaque appel récursif et on finit donc par atteindre le cas de base d'une liste ayant un seul élément : l'arbre résultat.

On notera que la récursion est terminale ainsi que l'appel récursif interne inserer.

```
let rec fusionner l = match l with
  | [] -> failwith "La liste doit être non vide !"
  | [a] -> a
  | a :: b :: suite -> fusionner (insérer (Noeud(a,b)) suite)
;;
fusionner : arbre list -> arbre = <fun>

let Huffman s = fusionner (trier (compter s));;
Huffman : char list -> arbre = <fun>
```

On notera que les trois fonctions intervenant sont récursives terminales.

Question 20.
Construction de la clé de codage associée à un arbre de codage.

Dans le programme suivant, on désigne par :
- reste la partie de l'arbre non encore parcourue,
- clef_prov la partie déjà constuite de la clef,
- parcours_prov la liste des branches parcourues pour arriver au noeud actuel (dans l'ordre inverse du parcours). C'est donc une liste de valeurs binaires.

```
let rec construire_aux clef_prov parcours_prov reste = match reste with
  | Vide -> failwith ``Pas d'arbre de codage vide !''
  | Feuille (v,_) -> (v,rev parcours_prov) :: clef_prov
  | Noeud (fg,fd) -> construire_aux clef_prov (O :: parcours_prov) fg
                   @ construire_aux clef_prov (I :: parcours_prov) fd
;;
construire_aux :
 (char * boolean list) list ->
 boolean list -> arbre -> (char * boolean list) list = <fun>

let construire = construire_aux [] [];;
construire : arbre -> (char * boolean list) list = <fun>
```

Vérifions cette fonction avec les deux arbres de l'exemple III.2 :

```
construire arbre1;;
- : (char * boolean list) list =
 [‘a‘, [O; O]; ‘b‘, [O; I; O]; ‘c‘, [O; I; I]; ‘d‘, [I]]

construire arbre2;;
- : (char * boolean list) list =
 [‘a‘, [O; O]; ‘b‘, [O; I]; ‘c‘, [I; O]; ‘d‘, [I; I]]
```

Question 21.
Reconstruction de l'arbre de codage associé à une clé de codage.

On commence par écrire une fonction inclure (x,l) a_prov qui inclut le caractère x de code l dans la partie déjà construite a_prov de l'arbre :

```
let rec inclure (x,l) a_prov = match (l, a_prov) with
  | ([], Vide)                -> Feuille (x,0)
  | ([], _)                   -> failwith "Erreur dans la clé de codage !"
  | (b :: suite, Noeud(fg,fd)) ->
      if b = O then Noeud((inclure (x,suite) fg), fd)
               else Noeud(fg, (inclure (x,suite) fd))
  | (b :: suite, a)           ->
      (* a est soit le Vide soit une Feuille *)
      if b = O then Noeud ((inclure (x,suite) a), Vide)
      else Noeud (Vide, inclure (x,suite) a)
;;
inclure : char * boolean list -> arbre -> arbre = <fun>
```


Puis on écrit une fonction reconstruire_aux qui "incorpore" clef-reste (partie non encore traitée de la clé de codage) à la partie a_prov déjà construite :

```
let rec reconstruire_aux clef_reste a_prov =
  match clef_reste with
    | []         -> a_prov
    | a :: suite -> reconstruire_aux suite (inclure a a_prov)
;;
reconstruire_aux : (char * boolean list) list -> arbre -> arbre = <fun>
```

Il ne reste plus qu'à écrire la fonction reconstruire elle-même :

```
let reconstruire clef = reconstruire_aux clef Vide;;
reconstruire : (char * boolean list) list -> arbre = <fun>
```

Vérifions en reconstruisant les 2 arbres de l'exemple (naturellement les occurences mises à part) :

```
reconstruire (construire arbre1);;
- : arbre =
 Noeud
  (Noeud (Feuille (‘a‘, 0), Noeud (Feuille (‘b‘, 0), Feuille (‘c‘, 0))),
   Feuille (‘d‘, 0))

reconstruire( construire arbre2);;
- : arbre =
 Noeud
  (Noeud (Feuille (‘a‘, 0), Feuille (‘b‘, 0)),
   Noeud (Feuille (‘c‘, 0), Feuille (‘d‘, 0)))
```

Question 22.
Décodage d'une suite binaire à l'aide d'un arbre de codage.

Dans la fonction suivante, s_reste désigne la partie non encore parcourue de la liste binaire, a_reste le noeud de l'arbre de codage où l'on se trouve et rev_texte_prov le miroir de la liste de caractères déjà construite.

On notera que cette fonction est récursive terminale.

```
let rec decoder_aux a s_reste a_reste rev_texte_prov =
  match (s_reste, a_reste) with
    | ([], Feuille (x,_))         -> x :: rev_texte_prov
    | ([], _)                     -> failwith "Erreur dans le codage !"
    | (b :: suite, Vide)          -> failwith "Arbre de codage vide !"
    | (b :: suite, Feuille (x,_)) ->
        decoder_aux a s_reste a (x :: rev_texte_prov)
    | (b :: suite, Noeud (fg,fd)) ->
        if b = O then decoder_aux a suite fg rev_texte_prov
                 else decoder_aux a suite fd rev_texte_prov
;;
decoder_aux : arbre -> boolean list -> arbre -> char list -> char list =
 <fun>

let decoder s a = rev (decoder_aux a s a []);;
decoder : boolean list -> arbre -> char list = <fun>
```

Vérifions que tout ce "bazar" fonctionne.

Afin que ce soit plus parlant commençons par écrire deux fonctions simplistes qui transforment une chaîne de caractères en une liste de caractères et réciproquement :

```
let liste_char_of_string texte =
  let n = string_length texte
  and liste = ref [] in
    for i=0 to n-1 do liste := (nth_char texte i) :: !liste done;
    rev !liste
;;
liste_char_of_string : string -> char list = <fun>

let rec string_of_liste_char suite = match suite with
  | [] -> ""
  | [c] -> make_string 1 c
  | c :: reste -> (make_string 1 c) ^ string_of_liste_char reste
;;
string_of_liste_char : char list -> string = <fun>
```

Puis vérifions :

```
let message_clair = "les sanglots longs des violonts de l'automne";;
message_clair : string = "les sanglots longs des violonts de l'automne"

let clef = construire (Huffman (liste_char_of_string message_clair));;
clef : (char * boolean list) list =
 ['a', [0; 0; 0; 0]; 'd', [0; 0; 0; 1]; 'm', [0; 0; 1; 0; 0];
  ''', [0; 0; 1; 0; 1]; 'g', [0; 0; 1; 1]; 'l', [0; 1; 0]; 'o', [0; 1; 1];
  'u', [1; 0; 0; 0; 0]; 'v', [1; 0; 0; 0; 1; 0]; 'i', [1; 0; 0; 0; 1; 1];
  't', [1; 0; 0; 1]; 's', [1; 0; 1]; ' ', [1; 1; 0]; 'e', [1; 1; 1; 0];
  'n', [1; 1; 1; 1]]

let message_codé = coder (liste_char_of_string message_clair) clef;;
message_codé : boolean list =
 [0; 1; 0; 1; 1; 1; 0; 1; 0; 1; 1; 1; 0; 1; 0; 1; 0; 0; 0; 0; 1; 1; 1; 1; 0;
  0; 1; 1; 0; 1; 0; 0; 0; 1; 1; 1; 0; 0; 1; 1; 0; 1; 1; 1; 0; 0; 1; 0; 0; 1; 1;
  1; 1; 1; 1; 0; 0; 1; 1; 1; 0; 1; 1; 1; 0; 0; 0; 0; 1; 1; 1; 1; 0; 1; 0; 1;
  1; 1; 0; 1; 0; 0; 0; 1; 0; 1; 0; 0; 0; 1; 1; 0; 1; 1; 0; 1; 0; 0; 1; 1;
  ...]

let message_décodé = string_of_liste_char(
                     decoder message_codé (reconstruire clef));;
message_décodé : string = "les sanglots longs des violonts de l'automne"
```


Question 23.

Pour construire un arbre de Huffman minimal, on commence par construire la liste non triée des feuilles (caractère,occurence). Puis on trie cette liste par insertion d'où un coût de $O(p^2)$ comparaisons : au pire $\frac{p(p-1)}{2}$ comparaisons soit $p(p-1)$ appels à la fonction nombre.

Puis on appelle $p-1$ fois la fonction `fusionner` qui elle-même appelle la fonction `insérer` qui fait au pire q comparaisons (en notant q la longueur de la liste dans laquelle on insère). Donc au pire `fusionner` fait encore $\frac{p(p-1)}{2}$ comparaisons.

Ainsi au total la fonction la plus appelée est la fonction nombre pouvant être appelée $2p^2$ fois. □

Questions 24 et 25.

Il y a plusieurs possibilités d'améliorer la complexité temporelle de la construction d'un arbre (minimal) de Huffman associé à une suite de caractères.

Tout d'abord, sans diminuer le nombre de comparaisons, on peut modifier la structure de donnée arbre de manière à ce que le coût d'une comparaison soit plus faible. Pour cela il suffit que chaque noeud interne comporte une étiquette indiquant le nombre total d'occurences du sous-arbre dont il est racine.
`type arbre = Vide | Feuille of char * int | Noeud of int * arbre * arbre;;`
Cela compliquerait assez peu la fonction `Huffman` et la comparaison de deux arbres se ferait alors en temps constant en lisant l'étiquette des racines au lieu de faire appel à la fonction nombre qui doit parcourir chaque arbre !
L'amélioration est nulle pour le tri de la liste des feuilles mais très importante dans la fonction `fusionner` utilisée un grand nombre de fois par la fonction `Huffman` avec des arbres de taille de plus en plus grande.

Ensuite pour trier la liste "anarchique"" des feuilles, au lieu d'utiliser un tri par insertion, on peut utiliser un tri par fusion (facile à mettre en œuvre pour les listes). D'où un coût seulement en $O(n \ln n)$ comparaisons.

Pour l'insertion utilisée par `fusionner` dans `Huffman`, on pourrait faire une insertion dichotomique. Certes la complexité en parcours de la liste reste linéaire mais le nombre de comparaisons passe en $O(\ln k)$ pour l'insertion dans une liste de longueur k. Le coût en comparaisons de `Huffman` est alors de $O(\ln(n!)$ soit (par la formule de Stirling) de $O(n \ln n)$.
Enfin on peut éviter le tri de la liste anarchique des feuilles en la construisant de manière à ce qu'elle soit triée d'emblée.

Première version.

Tout d'abord on écrit une fonction `occurences v l` qui renvoie le nombre d'occurences du caractère `v` dans la liste de feuilles `l` si `v` y figure et 0 sinon.
Cette fonction est récursive terminale.

```
#let rec occurences v l = match l with
  | [] -> 0
  | Feuille (a,p) :: suite -> if a = v then p
                              else occurences v suite
  | _ -> failwith "Liste de feuilles uniquement !"
;;
occurences : char -> arbre list -> int = <fun>
```

Puis on écrit la fonction ajouter2 dont le rôle est décrit dans la question 25 :

```
let rec ajouter2 v l =
  let p = occurences v l in
  match l with
    | []                                  -> [Feuille (v,1)]
    | Feuille (x,k) :: suite when k > p+1  -> Feuille (v,p+1) :: l
    | Feuille (x,k) :: suite when (x <> v) ->
       (* on a forcément k <= p+1 *)
       Feuille (x,k) :: (ajouter2 v suite)
    | Feuille (v,p) :: suite               ->
      begin
       match suite with
         | []                       -> [Feuille (v,p+1)]
         | Feuille (y,m) :: ssuite -> if m = p
             then Feuille (y,m) :: (ajouter2 v (Feuille (v,p) :: ssuite))
             else Feuille (v,p+1) :: suite
         | _                        -> failwith "Liste de feuilles !"
      end
    | _                                   -> failwith "Liste de feuilles !"
;;
ajouter2 : char -> arbre list -> arbre list = <fun>
```

Cette version présente l'inconvénient (minime) de parcourir 2 fois la liste : déjà pour déterminer la présence et l'occurence éventuelle du caractère v dans la liste puis pur la fonction elle-même.

Deuxième version.

La version suivante ne parcourt la liste qu'une seule fois :

```
let rec ajouter3_aux v l = match l with
  | []                     -> (false, [])
  | Feuille (x,k) :: suite ->
      if x = v then (true, insérer (Feuille (x,k+1)) suite)
      else let (présent, ll) = ajouter3_aux v suite in
        (présent, Feuille (x,k) :: ll)
  | _                      -> failwith "Liste de feuilles !"
;;
ajouter3_aux : char -> arbre list -> bool * arbre list = <fun>

let ajouter3 v l =
  let (présent, ll) = ajouter3_aux v l in
    if présent then ll else Feuille (v,1) :: ll
;;
ajouter3 : char -> arbre list -> arbre list = <fun>
```

Vérification sur un exemple :

```
let rec liste_feuilles l = match l with
  | [] -> []
  | c :: suite -> ajouter3 c (liste_feuilles suite)
;;
liste_feuilles : char list -> arbre list = <fun>

let message_clair = "les sanglots longs des violonts de l'automne";;
message_clair : string = "les sanglots longs des violonts de l'automne"

liste_feuilles (liste_char_of_string message_clair);;
- : arbre list =
 [Feuille (‘m‘, 1); Feuille (‘u‘, 1); Feuille (‘’‘, 1); Feuille (‘i‘, 1);
  Feuille (‘v‘, 1); Feuille (‘a‘, 2); Feuille (‘g‘, 2); Feuille (‘d‘, 2);
  Feuille (‘t‘, 3); Feuille (‘e‘, 4); Feuille (‘n‘, 4); Feuille (‘l‘, 5);
  Feuille (‘o‘, 5); Feuille (‘s‘, 6); Feuille (‘ ‘, 6)]
```

FIN