Corrigé : codage de Huffman

Partie I. Codage d'une suite de caractères

Question 1.

a) Commençons par écrire une fonction **assoc** de type *char -> clef -> code* telle que **assoc u c** renvoie le codage du caractère u selon le code c :

```
let rec assoc u = function
  | []                    -> raise Not_found
  | (v, b)::_ when u = v -> b
  | _::q                  -> assoc u q ;;
```

On peut noter que cette fonction est prédéfinie en Caml.
On définit ensuite :

```
let rec coder s c = match s with
  | []   -> []
  | u::q -> (assoc u c) @ (coder q c) ;;
```

ou de manière équivalente :

```
let coder s c = list_it (fun a b -> (assoc a c) @ b) s [] ;;
```

b) La fonction **assoc** réalise un parcours linéaire de la liste c de longueur p, chaque opération réalisée étant de coût constant. Sa complexité est donc en $O(p)$.
La fonction **coder** réalise un parcours linéaire de la liste s en réalisant deux opérations : un calcul de la fonction **assoc** de complexité $O(p)$ et une concaténation de complexité $O(|c(s_i)|)$, pour $1 \leqslant i \leqslant n$. La complexité totale est donc en :

$$\sum_{i=1}^{n} O(p+|c(s_i)|) = O\Big(np + \sum_{u\in\Sigma} |c(u)| \times \mathrm{Occ}_s(u)\Big) = O(np + C(s)).$$

c) De l'égalité $C(s) = \sum_{i=1}^{n} |c(s_i)|$ il résulte immédiatement que $C(s)$ n'est autre que la longueur du codage de s selon c.

```
let cout s c = list_length (coder s c) ;;
```

Question 2.

a)

```
let rec prefixe b1 b2 = match (b1, b2) with
    | [], _      -> true
    | _, []      -> true
    | u::q, v::r -> u = v && prefixe q r ;;
```

b)

```
let separable c =
  let rec aux = function
    | []   -> true
    | b::q -> not (exists (prefixe b) q) && aux q
  in aux (map snd c) ;;
```

La fonction **aux** vérifie si dans une liste de codes il en existe deux dont l'un est préfixe de l'autre ; il suffit alors d'appliquer cette fonction à la liste des secondes composantes des éléments de c.

c) La fonction **`prefixe`** réalise un parcours partiel des deux listes passées en argument donc on peut majorer sa complexité par un $O(m)$. Le calcul de **`exists (prefixe b) q`** a donc une complexité en $O(m\ell)$ où ℓ est la taille de la liste **`q`**, quantité que l'on peut majorer par un $O(mp)$. Cette opération est répétée p fois, donc la complexité de la fonction **`aux`** peut être majorée par un $O(mp^2)$. La complexité du calcul de **`map snd c`** est en $O(p)$, donc la complexité de la fonction **`separable`** est en $O(mp^2)$.

Partie II. Arbre de Huffman

Question 3.

a) Supposons qu'un nœud interne possède une étiquette u. Il possède au moins un descendant qui est une feuille, étiquetée par un mot v. Le chemin qui mène de la racine à la feuille étiquetée par v passe par le nœud étiqueté par u, donc $c(v)$ est préfixe de $c(u)$ et c n'est pas séparable.
Réciproquement, si aucun nœud interne n'est étiqueté, considérons deux mots distincts u et v. Ceux-ci sont les étiquettes de deux feuilles distinctes ; considérons le plus proche ancêtre commun de ces deux feuilles ; le chemin qui y mène est étiqueté par la suite de bits $b_1b_2\cdots b_{k-1}$, et $c(u)$ et $c(v)$ s'écrivent respectivement $b_1b_2b_{k-1}b_k\cdots$ et $b_1b_2b_{k-1}b'_k\cdots$ avec $b'_k \neq b_k$, donc aucun n'est préfixe de l'autre.

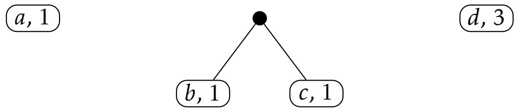
b) Soit u un caractère, $c(u) = b_1b_2\cdots b_r$ son code et $k \in [\![1,r]\!]$. Le chemin $b_1b_2\cdots b_{k-1}$ conduit à un nœud interne qui possède deux fils puisque a est un arbre binaire strict. L'un de ces deux fils est un ancêtre de u, l'autre possède au moins une feuille parmi ses descendants, et cette dernière est atteinte par un chemin de la forme $b'_1b'_2\cdots b'_s$ avec $b'_i = b_i$ pour $i < k$ et $b'_k \neq b_k$. La clef de codage c est bien optimale.

Question 4.

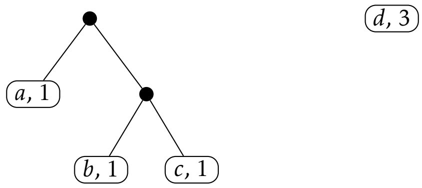
```
let rec nombre = function
  | Vide           -> 0
  | Feuille (_, n) -> n
  | Noeud (g, d)   -> nombre g + nombre d ;;
```

Question 5. On dispose au début de l'algorithme de 4 feuilles d'occurrences respectives 1, 1, 1, 3 :

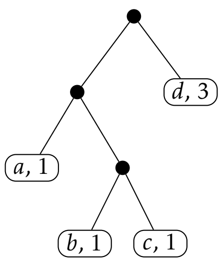
Deux de ces feuilles d'occurrences 1 sont fusionnées (n'importe lesquelles) :



On dispose maintenant de trois arbres d'occurrences respectives 1, 2 et 3. Les deux premiers sont fusionnés :



On dispose enfin de deux arbres d'occurrences 3, qui sont fusionnés :



Question 6.

a) u contribue au coût de H dans la proportion $\mathrm{C}(u) \times \mathrm{Occ}(u)$, $\mathrm{C}(u)$ désignant la profondeur de la feuille étiquetée par u dans H. On dispose d'une formule analogue pour v, et ainsi,

$$\Delta = (\mathrm{C}(v) - \mathrm{C}(u))\mathrm{Occ}(u) + (\mathrm{C}(u) - \mathrm{C}(v)\mathrm{Occ}(v) = (\mathrm{C}(v) - \mathrm{C}(u))(\mathrm{Occ}(u) - \mathrm{Occ}(v)).$$

b) Considérons dans un arbre minimal H une feuille de profondeur maximale $\mathrm{C}(u)$ étiquetée par u. S'il existait une feuille étiquetée par v telle que $\mathrm{C}(v) < \mathrm{C}(u)$ et $\mathrm{Occ}(v) < \mathrm{Occ}(u)$. La permutation de ces deux feuilles créerait un arbre H' tel que $\Delta < 0$, autrement dit tel que $\mathrm{C}(\mathrm{H}') < \mathrm{C}(\mathrm{H})$, ce qui contredirait le caractère minimal de H. Les feuilles de profondeur maximale sont donc étiquetées par les caractères d'occurrences minimales.

Question 7.

a) Notons h la profondeur des feuilles u et v de H ; celles-ci contribuent au coût de H par la quantité $h(\mathrm{Occ}(u) + \mathrm{Occ}(v))$. Posons $\mathrm{K} = \mathrm{C}(\mathrm{H}) - h(\mathrm{Occ}(u) + \mathrm{Occ}(v))$; cette quantité correspond à la contribution de tous les autres caractères au coût de H. On a alors $\mathrm{C}(\mathrm{H}') = \mathrm{K} + (h-1)(\mathrm{Occ}(u) + \mathrm{Occ}(v)) = \mathrm{C}(\mathrm{H}) - (\mathrm{Occ}(u) + \mathrm{Occ}(v))$.

Considérons maintenant un arbre de Huffman minimal A possédant les mêmes feuilles que H. D'après la question précédente, u et v se trouvent à la profondeur maximale de A (puisque l'arbre binaire est strict il y a au moins deux feuilles à cette profondeur) et sans modifier le coût on peut supposer, quitte à permuter deux feuilles de cette profondeur, qu'elles ont un même père n. Supprimons dans A ces deux feuilles et considérons que n est une feuille d'occurrence $\mathrm{Occ}(u) + \mathrm{Occ}(v)$. On obtient un arbre A', et puisque H' est minimal, $\mathrm{C}(\mathrm{A}') \geqslant \mathrm{C}(\mathrm{H}')$. Mais alors $\mathrm{C}(\mathrm{A}) = \mathrm{C}(\mathrm{A}') + \mathrm{Occ}(u) + \mathrm{Occ}(v) \geqslant \mathrm{C}(\mathrm{H}') + \mathrm{Occ}(u) + \mathrm{Occ}(v) = \mathrm{C}(\mathrm{H})$, et puisque A est minimal il en est de même de H.

b) Raisonnons par récurrence sur le nombre de caractères p (qui est aussi le nombre de feuilles de l'arbre de Huffman).

- Lorsque $p = 1$ le résultat est évident : l'arbre obtenu par l'algorithme de Huffman se réduit à une feuille.
- Si $p > 1$, considérons la première fusion réalisée par l'algorithme de Huffman : elle concerne deux caractères u et v d'occurrences minimales, qui deviennent les fils d'un même nœud n. Considérons maintenant l'alphabet dans lequel les caractères u et v ont été remplacés par le caractère n, avec $\mathrm{Occ}(n) = \mathrm{Occ}(u) + \mathrm{Occ}(v)$. cet alphabet ne contient plus que $p - 1$ caractères, et par hypothèse de récurrence la suite de l'algorithme construit un arbre de Huffman minimal H' pour ce nouvel alphabet. En remplaçant H' par H la question précédente a montré que H est aussi minimal pour l'alphabet Σ, ce qui permet de conclure.

Partie III. Tri par insertion

Question 8.

```
let comparer a1 a2 = nombre a1 < nombre a2 ;;
```

Question 9.

```
let rec inserer a = function
  | t::q when comparer t a -> t::(inserer a q)
  | l                      -> a::l ;;
```

Question 10. Il suffit de trier récursivement la queue de la liste puis d'insérer la tête :

```
let rec trier = function
  | []   -> []
  | t::q -> inserer t (trier q) ;;
```

ou si on préfère utiliser une fonctionnelle :

```
let trier lst = list_it inserer lst [] ;;
```

Partie IV. Construction des codes de Huffman

Question 11.

```
let rec ajouter u = function
  | []                         -> [Feuille (u, 1)]
  | Feuille (v, n)::q when u = v -> Feuille (u, n+1)::q
  | t::q                       -> t::(ajouter u q) ;;
```

Question 12.

```
let rec compter = function
  | []   -> []
  | u::q -> ajouter u (compter q) ;;
```

ou avec une fonctionnelle :

```
let compter lst = list_it ajouter lst [] ;;
```

Question 13. On applique l'algorithme de Huffman : si la liste contient au moins deux arbres, les deux premiers (d'occurrences minimales) sont fusionnés puis réinsérés en respectant l'invariance du tri par occurrence croissante.

```
let rec fusionner = function
  | []      -> Vide
  | [a]     -> a
  | a::b::q -> fusionner (inserer (Noeud (a, b)) q) ;;
```

Question 14. Il reste à assembler les fonctions précédentes :

```
let huffman s = fusionner (trier (compter s)) ;;
```

Partie V. Codage et décodage d'une suite

Question 15. Nous allons réaliser un parcours en profondeur de l'arbre en accumulant une suite de bits correspondant au parcours réalisé. À chaque fois qu'une feuille est atteinte, l'accumulateur contient l'image miroir du code du caractère attaché à cette feuille.

```
let construire a =
  let rec aux clef chemin = function
    | Vide           -> failwith "construire"
    | Feuille (u, _) -> (u, rev chemin)::clef
    | Noeud (g, d)   -> aux (aux clef (false::chemin) g) (true::chemin) d
  in aux [] [] a ;;
```

L'accumulateur **`clef`** contient la liste des codes déjà trouvés et **`chemin`** le chemin menant de la racine au nœud, avec pour convention **`false`** pour un déplacement vers le fils gauche et **`true`** vers le fils droit.

Question 16. Le principe général de l'algorithme est, partant de l'arbre vide, d'y insérer progressivement chacune des feuilles en construisant si besoin est le chemin qui y mène.

```
let reconstruire c =
  let rec aux (u, b) a = match (b, a) with
      | [], Vide               -> Feuille (u, 0)
      | false::q, Noeud (g, d) -> Noeud (aux (u,q) g, d)
      | true::q, Noeud (g, d)  -> Noeud (g, aux (u, q) d)
      | false::q, _            -> Noeud (aux (u, q) Vide, a)
      | true::q, _             -> Noeud (a, aux (u, q) Vide)
      | _                      -> failwith "reconstruire"
  in list_it aux c Vide  ;;
```

La fonction **aux** prend en arguments un caractère et son code (u, b) ainsi que l'arbre a en cours de construction. On commence par suivre le chemin décrit par le code b (lignes 4 et 5). Il arrive un moment où l'on tombe sur une feuille ou l'arbre vide (lignes 6 et 7) ; on poursuit alors en ajoutant des nœuds. Vient enfin le moment où on arrive à l'extrémité du chemin (ligne 3). On y insère alors la feuille souhaitée.

Question 17. Les bits contenus dans b décrivent des chemins que l'on suit dans a ; chaque feuille atteinte donne un nouveau caractère décodé ; on recommence ensuite à la racine.

```
let decoder b a =
  let rec aux l t = match l, t with
    | _, Feuille (u, _)      -> u::(aux l a)
    | [], _                  -> []
    | false::q, Noeud (g, _) -> aux q g
    | true::q, Noeud (_, d)  -> aux q d
    | _                      -> failwith "decoder"
in aux b a ;;
```

C'est la fonction **aux** qui réalise le parcours dans l'arbre décrit par la suite de bits (lignes 5 et 6). Chaque feuille atteinte (ligne 3) fournit un caractère et recommence un parcours à partir de la racine de l'arbre a, jusqu'à avoir lu tous les bits (ligne 4).