Corrigé : dictionnaires (X 2006)

Partie I. Mots

Question 1. On définit la fonction suivante :

```
let imprimer mot =
  let rec aux = function
    | [] -> ()
    | t::q -> aux q ; lettre t
  in aux ((-1)::mot) ;;
```

Question 2. Il s'agit ici de redéfinir la fonction `rev` :

```
let inverseDe = it_list (fun a b -> b::a) [] ;;
```

Partie II. Dictionnaires

Question 3.

a) À chaque nœud est associé un vecteur de N + 1 cases : les cases indicées entre 1 et N représentent un caractère de l'alphabet $\mathcal{A}$, la case d'indice 0 le caractère \$.
Ainsi, le dictionnaire vide est représenté par l'arbre : `NoeudT v` où `v` est un vecteur de N + 1 cases contenant l'arbre `VideT` dans toutes ses cases et le dictionnaire contenant l'unique mot vide par un arbre : `NoeudT w` où `w` est un vecteur de N + 1 cases contenant l'arbre `VideT` dans les cases indicées entre 1 et N et la représentation du dictionnaire vide dans la case d'indice 0. On notera qu'avec un tel choix, `VideT` ne représente pas un dictionnaire.

Figure 1 – Représentations tabulées du dictionnaire vide et du dictionnaire contenant le mot vide.

b) Dans un sens le résultat demandé est immédiat : par définition, dans un dictionnaire tabulé les branches étiquetées par \$ sont les branches qui mènent à une feuille.

Réciproquement, considérons un arbre possédant cette propriété de cohérence et montrons par induction structurelle que cet arbre représente un dictionnaire.

- C'est bien le cas de l'arbre représentant le dictionnaire vide, dont la racine n'a pas de fils.
- Considérons maintenant un arbre vérifiant la propriété de cohérence et dont la racine possède au moins un fils. Chacun de ces fils vérifie la propriété de cohérence donc par hypothèse d'induction représente un dictionnaire. Notons $\mathcal{D}_1, \dots, \mathcal{D}_k$ ceux-ci, et notons $a_1, \dots, a_k$ les étiquettes des branches qui y mènent.
  - Si aucune de ces étiquettes n'est le caractère \$, l'arbre représente le dictionnaire $a_1\mathcal{D}_1 \cup \dots \cup a_k\mathcal{D}_k$;
  - si l'une de ces étiquettes, par exemple a_1, est égale à \$, d'après la propriété de cohérence, $\mathcal{D}_1$ est le dictionnaire vide et l'arbre représente le dictionnaire $\{\varepsilon\} \cup a_2\mathcal{D}_2 \cup \dots \cup a_k\mathcal{D}_k$ où ε désigne le mot vide.

Question 4. On utilise une fonction auxiliaire et un accumulateur dans lequel sont stockées les lettres utilisées pour former les mots du dictionnaire. On imprime l'accumulateur à chaque fois qu'on aboutit à une feuille.

```
let imprimerDictTab d =
  let rec aux acc = function
    | VideT        -> ()
    | NoeudT v     -> if v.(0) <> VideT then imprimer acc ;
                      for i = 1 to N do aux (i::acc) v.(i) done
  in aux [] d ;;
```

Question 5. Un mot est dans un dictionnaire s'il conduit à une feuille :

```
let rec estDansDictTab u d = match u, d with
  | _, VideT       -> failwith "estDabsDictTab"
  | [], NoeudT v   -> v.(0) <> VideT
  | t::q, NoeudT v -> v.(t) <> VideT && estDansDictTab q v.(t) ;;
```

Le premier motif a pour unique objet de rendre le filtrage exhaustif puisque l'arbre **`VideT`** ne représente aucun dictionnaire. On peut bien entendu s'en passer.

Question 6. On procède peu ou prou comme pour la fonction précédente, en construisant une nouvelle branche là où il n'en existe pas encore.

```
let rec ajoutADictTab u d = match u, d with
  | _, VideT                          -> failwith "ajoutDictTab"
  | [], NoeudT v                      -> let w = make_vect (N+1) VideT in
                                         v.(0) <- NoeudT w ; NoeudT v
  | t::q, NoeudT v when v.(t) = VideT -> let w = make_vect (N+1) VideT in
                                         v.(t) <- ajoutADictTab q (NoeudT w) ; NoeudT v
  | t::q, NoeudT v                    -> v.(t) <- ajoutADictTab q v.(t) ; NoeudT v ;;
```

Remarque. On peut améliorer le coût spatial de cette fonction en déclarant une constante globale représentant le dictionnaire vide :

```
let dictVide = NoeudT make_vect (N+1) VideT ;;
```

et en remplaçant le deuxième motif du filtrage par :

```
| [], NoeudT v -> v.(0) <- dictVide ; NoeudT v
```

En effet, dans la fonction ci-dessus chaque dictionnaire vide ajouté pour marquer la fin de mot possède son propre espace mémoire, mutable, bien qu'il ne soit jamais modifié compte tenu de la propriété de cohérence. Avec la modification suggérée cet inconvénient disparait : le même espace mémoire est utilisé pour marquer toutes les fins de mot.

Partie III. Dictionnaire binaire

Question 7. La racine n'ayant pas de frère ne possède qu'un fils gauche correspondant au premier fils ; ne portant pas d'information utile il est donc inutile de la représenter, et la nouvelle racine devient le premier des fils de l'arbre tabulé. De ce fait, le dictionnaire vide est maintenant représenté par l'arbre binaire **`VideB`** et le dictionnaire contenant uniquement le mot vide par l'arbre binaire **`NoeudB (VideB, 0, VideB)`** (soit un arbre réduit à sa racine, étiquetée par le caractère \$).

Question 8. Même démarche qu'à la question 4 : on utilise un accumulateur pour stocker les lettres rencontrées.

```
let imprimerDictBin d =
  let rec aux acc = function
    | VideB              -> ()
    | NoeudB (_, 0, fd)  -> imprimer acc ; aux acc fd
    | NoeudB (fg, c, fd) -> aux (c::acc) fg ; aux acc fd
  in aux [] d ;;
```

On peut observer qu'un nœud étiqueté par la lettre \$ ne peut avoir de fils gauche (c'est la propriété de cohérence pour cette représentation).

Question 9.

```
let rec estDansDictBin u d = match u, d with
  | _, VideB                -> false
  | [], NoeudB (_, c, _)    -> c = 0
  | t::q, NoeudB (fg, c, fd) -> (t = c && estDansDictBin q fg) || (estDansDictBin u fd) ;;
```

Question 10.

```
let rec ajoutADictBin u d = match u, d with
  | [], VideB                          -> NoeudB (VideB, 0, VideB)
  | [], NoeudB (_, 0, _)               -> d
  | t::q, VideB                        -> NoeudB (ajoutADictBin q VideB, t, VideB)
  | t::q, NoeudB (fg, c, fd) when c = t -> NoeudB (ajoutADictBin q fg, t, fd)
  | _, NoeudB (fg, c, fd)              -> NoeudB (fg, c, ajoutADictBin u fd) ;;
```

Partie IV. Comparaison des coûts ; conversion de représentations

Question 11.

a) Dans chacune des deux représentations le nombre S de sommets est identique à une unité près, correspondant à la suppression de la racine pour la représentation binaire. Je choisis d'appeler S le nombre de sommets pour la représentation binaire.

Notons d le dictionnaire et observons la structure binaire : il y a autant de nœuds étiquetés par le caractère \$ que de mots dans le dictionnaire ; on peut donc affirmer que $S \geqslant \sum_{u\in d} 1 = |d|$. Par ailleurs, chaque lettre qui étiquète un nœud appartient à au moins un mot de d. On a donc $S \leqslant \sum_{u\in d}(|u|+1) = |d|\Big(1 + \frac{1}{|d|}\sum_{u\in d}|u|\Big)$.

Compte tenu des hypothèses faites, on a $|d| = n^5$ et $\frac{1}{|d|}\sum_{u\in d}|u| = n$ soit : $n^5 \leqslant S \leqslant n^6 + n^5$.

En conclusion, on peut donc affirmer que $S = O(n^6)$.

Dans le cas d'une représentation tabulée, chaque nœud est représenté par un tableau de taille N pour une complexité spatiale en O(N) soit en tout en $O(NS) = O(Nn^6)$. Pour une représentation binaire, chaque nœud est représenté par un emplacement mémoire de taille fixe pour une complexité totale en $O(S) = O(n^6)$. Avantage donc à la représentation binaire.

b) Pour le test d'appartenance d'un mot de taille ℓ (ou l'ajout, les deux fonctions suivent des démarches voisines pour un coût temporel identique) dans un dictionnaire tabulé, le choix de la bifurcation dans l'arbre tabulé se fait en coût constant (accès à un élément d'un tableau connu par son indice) donc la complexité temporelle est en $O(\ell)$.

Pour un dictionnaire représenté par un arbre binaire la descente est plus longue puisqu'il faut parcourir tout ou partie des frères d'un nœud avant de passer à la lettre suivante. La complexité pour avancer d'une lettre est donc en O(N), pour une complexité totale en $O(\ell N)$. Cette fois-ci, l'avantage va à la représentation tabulée des dictionnaires.

Question 12.

```
let rec tabVersBin d =
  let rec aux v = function
    | k when k = N + 1      -> VideB
    | 0 when v.(0) <> VideT -> NoeudB (VideB, 0, aux v 1)
    | k when v.(k) <> VideT -> NoeudB (tabVersBin v.(k), k, aux v (k+1))
    | k                     -> aux v (k+1)
  in match d with
       | VideT -> failwith "tabVersBin"
       | NoeudT v -> aux v 0 ;;
```

La fonction auxiliaire prend pour arguments un vecteur constitué des fils d'un nœud de l'arbre ainsi qu'un indice $k \in [\![0, N+1]\!]$ pour parcourir ce vecteur à la recherche de cases non vides. La valeur $k = 0$ est traitée à part compte tenu de la propriété de cohérence (elle marque la fin d'un mot).

Question 13.

```
let rec binVersTab = function
  | VideB                              -> NoeudT (make_vect (N+1) VideT)
  | NoeudB (VideB, c, fd) when c <> 0 -> binVersTab fd
  | NoeudB (fg, c, fd)                 -> match binVersTab fd with
                                            | VideT    -> failwith "binVersTab"
                                            | NoeudT v -> v.(c) <- binVersTab fg ; NoeudT v ;;
```

On peut faire la même remarque que celle faite à la question 6 : il est préférable de définir une variable globale représentant le dictionnaire tabulé vide et de remplacer le premier motif par :

```
  | VideB -> dictVide
```

Partie V. Le mot le plus long

Question 14. On utilise une fonction qui extrait une lettre d'un mot :

```
let rec extrait c = function
  | []              -> raise Not_found
  | t::q when t = c -> q
  | t::q            -> t::(extrait c q) ;;
```

Cette fonction est de type *int -> mot -> mot*.
Si on utilise la représentation binaire, on écrit ensuite :

```
let imprimerMotsDans d u =
  let rec aux acc u = function
    | VideB                          -> ()
    | NoeudB (_, 0, fd)              -> imprimer acc ; aux acc u fd
    | NoeudB (fg, c, fd) when mem c u -> aux (c::acc) (extrait c u) fg ; aux acc u fd
    | NoeudB (fg, c, fd)             -> aux acc u fd
  in aux [] u d ;;
```

L'accumulateur transporte la liste des lettres sélectionnées et **u** la liste des lettres disponibles.

Notons $C(|u|,|d|)$ la complexité de cette fonction, où $|u|$ désigne la longueur de u et $|d|$ le nombre de nœuds du dictionnaires. Si $d =$ **NoeudB** $(f_g,\ c,\ f_d)$ on dispose de la relation :

$$C(|u|,|d|) \leqslant O(u) + C(|u|-1,|f_g|) + C(|u|,|f_d|)$$

car les fonctions **mem** et **extrait** ont une complexité en $O(|u|)$.
Il existe un entier B tel que $C(|u|,|d|) \leqslant B.|u| + C(|u|-1,|f_g|) + C(|u|,|f_d|)$; considérons un entier $M > B$ et montrons par induction que $C(|u|,|d|) \leqslant M.|u|.|d|$.
En effet, si on suppose le résultat acquis pour les dictionnaires représentés par f_g et f_d alors :

$$C(|u|,|d|) \leqslant B.|u| + M.(|u|-1).|f_g| + M|u|.|f_d| = B.|u| + M.|u|(|d|-1) - M.|f_g| \leqslant B.|u| + M.|u|.|d| - M.|u| \leqslant M.|u|.|d|$$

On peut donc estimer la complexité de cette fonction en $O(|u|.|d|)$ soit encore un $O(|u|n^6)$ si on reprend les hypothèses de la partie IV.

Remarque. Si on préfère la représentation tabulée on obtient la version équivalente :

```
let imprimerMotsDans1 d u =
  let rec aux acc u = function
    | VideT    -> ()
    | NoeudT v -> if v.(0) <> VideT then imprimer acc ;
                  for k = 1 to N do
                    if v.(k) <> VideT && mem k u then aux (k::acc) (extrait k u) v.(k) ;
                  done
  in aux [] u d ;;
```

L'étude de la complexité est ici laissée au lecteur.

Partie VI. Anagrammes

Question 15. La démarche à suivre est somme toute assez proche de la question précédente, à deux exceptions près : l'impression ne se produit que lorsque toutes les lettres sont épuisées, et une fois un mot trouvé il faut poursuivre la recherche en insérant un espace dans l'accumulateur et en recommençant la recherche avec les lettres restantes et le dictionnaire initial.

```
let imprimerAnagrammes d u =
  let rec aux acc u = function
    | VideB                          -> ()
    | NoeudB (_, 0, fd) when u = []  -> imprimer acc
    | NoeudB (_, 0, fd)              -> aux acc u fd ; aux (0::acc) u d
    | NoeudB (fg, c, fd) when mem c u -> aux (c::acc) (extrait c u) fg ; aux acc u fd
    | NoeudB (fg, c, fd)             -> aux acc u fd
  in aux [] u d ;;
```

L'évaluation de la complexité est ici plus délicate du fait que la recherche recommence avec le dictionnaire initial à chaque fois qu'un mot est trouvé.

Pour la version tabulée on a :

```
let imprimerAnagrammes1 d u =
  let rec aux acc u = function
    | VideT     -> ()
    | NoeudT v when u = [] -> if v.(0) <> VideT then imprimer acc ;
    | NoeudT v  -> if v.(0) <> VideT then aux (0::acc) u d ;
                   for k = 1 to N do
                     if v.(k) <> VideT && mem k u then aux (k::acc) (extrait k u) v.(k) ;
                   done
  in aux [] u d ;;
```