X 2006 — Option informatique

Partie I — Mots

Question I.1
On redonne ici la fonction usuelle `do_list`.

```
type mot == int list ;;

let lettre = function
  | 0 -> print_char ' '
  | (-1) -> print_newline ()
  | i -> print_char (char_of_int (int_of_char 'a' - 1 + i)) ;;

let rec do_list f = function
  | [] -> ()
  | t :: q -> f t ; do_list f q ;;

let imprimer m =
  do_list lettre m ; lettre (-1) ;;
```

Programme 1 la fonction `imprimer`

Question I.2
Il s'agit de la fonction usuelle `rev`.

```
let inverseDe m =
  let rec miroir gd dg = match gd with
    | [] -> dg
    | t :: q -> miroir q (t :: dg)
  in
    miroir m [] ;;
```

Programme 2 la fonction `inverseDe`

Partie II — Dictionnaires

Question II.3
On choisit en pratique $N = 27$: les caractères usuels sont numérotés de 1 à 26, l'espace par 0 et le symbole `$` par $N = 27$.

II.3.a Le dictionnaire vide est représentée par `VideT`, alors que le dictionnaire contenant le seul mot vide est représenté par `NoeudT n` où `n` est un tableau dont tous les éléments contiennent `VideT`, sauf son élément d'indice N (qui correspond à `$`) : ce dernier est `NoeudT v` où cette fois `v` est un tableau dont tous les éléments contiennent `VideT`.

II.3.b Dans la définition même des *tries*, il est indiqué que *les branches qui mènent à une feuille sont étiquettées par* `$`et que ce sont les seules dans ce cas.
Inversement, si on considère une valeur de type `dictTab` qui vérifie la propriété de cohérence, elle représentera en effet un dictionnaire tabulé, dont l'ensemble des mots sont les étiquettes des chemins qui mènent de sa racine à une feuille, étiquettes privées du symbole `$` final.
Autrement dit, on interdit qu'un nœud de l'arbre auquel on accède en suivant le sélecteur `$` soit autre chose qu'une feuille, c'est-à-dire un nœud dont tous les descendants valent `VideT`.

Question II.4
On propose d'écrire une fonction auxiliaire récursive `imprimePrefixe : mot -> dictTab -> unit` qui imprime les éléments d'un sous-arbre, et qui admet en premier argument l'étiquette du chemin qui mène de la racine de l'arbre complet au sous-arbre considéré. Ce suffixe est un mot écrit de droite à gauche, car c'est ainsi qu'il est naturel de le construire.
On remarquera que le test `if n.(N) <> VideT then ...` permet de repérer les feuilles de l'arbre : ce sont elles qui demandent l'appel effectif à la fonction `imprimer`.

```
17  type dictTab =
18    | VideT
19    | NoeudT of dictTab vect ;;

20  let N = 27 ;;

21  (* les caractères de l'alphabet sont numérotés de 1 à N-1, $ est numéroté N *)
22  let imprimerDictTab d =
23    let rec imprimePrefixe prefixeDG = function
24      | VideT -> ()
25      | NoeudT n ->
26         for i=0 to N-1 do
27           imprimePrefixe (i::prefixeDG) n.(i)
28         done ;
29         (* n.(N) représente le noeud accessible par $ *)
30         if n.(N) <> VideT then imprimer (inverseDe prefixeDG)
31    in
32      imprimePrefixe [] d ;;
```

Programme 3 la fonction `imprimerDictTab`

Question II.5
La fonction `estDansDictTab` répond au même schéma : un mot est dans le dictionnaire s'il conduit à une feuille (et c'est ce que teste la ligne 36).

```
33  let rec estDansDictTab m = function
34    | VideT -> false
35    | NoeudT n -> match m with
36        | [] -> n.(N) <> VideT
37        | t :: q -> estDansDictTab q n.(t) ;;
```

Programme 4 la fonction `estDansDictTab`

Question II.6
L'ajout est plus complexe. On descend dans la structure jusqu'à ajouter le nouveau mot, en construisant des mots vides si la branche utile n'existe pas encore. Il faut faire attention au moment où l'on arrive à la feuille correspondante au mot inséré, et se rappeler la convention utilisée pour représenter les feuilles. On obtient le programme 5, page 3.

Partie III — Dictionnaire binaire

Question III.7
La racine de l'arbre construit n'aura pas de fils droit, puisqu'elle n'a pas de frère ! ce nœud, qui ne possède qu'un fils gauche et qui ne porte aucune information utile peut donc être supprimé.

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

39 let rec ajoutADictTab m d = match d with
40   | VideT -> let n = make_vect (N+1) VideT in
41       ( match m with
42          | [] -> n.(N) <- noeudVide ()
43          | t :: q -> n.(t) <- ajoutADictTab q (noeudVide ()) ) ;
44       NoeudT n
45   | NoeudT n ->
46       ( match m with
47          | [] -> if n.(N) = VideT then n.(N) <- noeudVide ()
48          | t :: q -> n.(t) <- ajoutADictTab q n.(t) ) ;
49       d ;;
```

Programme 5 la fonction `ajoutADictTab`

Question III.8
On procède de façon analogue à ce qu'on avait fait pour écrire la fonction `imprimerDictTab`. Mais ici, le test pour savoir si on arrive à une feuille (ligne 36 du programme 4) consiste tout simplement à consulter l'étiquette du nœud (ligne 53 du programme 6). On observera qu'un tel nœud ne peut avoir de fils gauche : c'est la propriété de cohérence du dictionnaire tabulé d'origine.

```
50 let imprimerDictBin d =
51   let rec imprimePrefixe prefixeDG = function
52     | VideB -> ()
53     | NoeudB(VideB,x,d) when x = N ->
54        imprimer (inverseDe prefixeDG) ;
55        imprimePrefixe prefixeDG d
56     | NoeudB(g,x,d) ->
57        imprimePrefixe (x :: prefixeDG) g ;
58        imprimePrefixe prefixeDG d
59   in
60     imprimePrefixe [] d ;;
```

Programme 6 la fonction `imprimerDictBin`

Question III.9
Le test étant simplifié, il est également plus facile d'écrire `estDansDictBin`.

```
61 let rec estDansDictBin m = function
62   | VideB -> false
63   | NoeudB(g,x,d) -> match m with
64       | [] -> x = N
65       | t :: q -> if t = x then estDansDictBin q g else estDansDictBin m d ;;
```

Programme 7 la fonction `estDansDictBin`

Toutefois on aura été attentif à ne rechercher que `q` quand on descend à gauche (on descend réellement dans le dictionnaire tabulé) mais à rechercher `m` tout entier quand on va à droite (on reste à la même hauteur dans le dictionnaire tabulé, et on consulte les frères du nœud courant).

Question III.10
L'ajout d'un mot au dictionnaire consiste à descendre dans l'arbre en descendant à droite si on n'a pas trouvé la lettre initiale (on cherche alors le mot entier parmi les frères) ou à gauche dans le cas contraire (on cherche alors le mot privé de son initiale dans l'arbre fils).
(Remarque : on aurait pu supprimer la ligne 72, que saurait gérer la ligne 74, mais il a semblé plus clair de détailler le filtrage.)

```
66  let rec ajoutADictBin m = function
67    | VideB -> ( match m with
68                 | [] -> NoeudB(VideB,N,VideB)
69                 | t :: q -> NoeudB(ajoutADictBin q VideB,t,VideB) )
70    | NoeudB(g,x,d) -> ( match m with
71                           | [] when x = N -> NoeudB(g,x,d)
72                           | [] -> NoeudB(g,x,ajoutADictBin m d)
73                           | t :: q when x = t -> NoeudB(ajoutADictBin q g,x,d)
74                           | _ -> NoeudB(g,x,ajoutADictBin m d) ) ;;
```

Programme 8 la fonction `ajoutADictBin`

Partie IV — Comparaison des coûts ; conversion de représentations

À titre de comparaison, on rappelle qu'un dictionnaire usuel de la langue française compte environ soixante mille mots, donc environ deux cent mille flexions. On pourra donc envisager des valeurs de l'ordre de $n \approx 12$.

Question IV.11
IV.11.a Les deux structures utilisent le même nombre S de sommets (à une unité près).
Ce nombre de sommets est évidemment majorés par la somme des longueurs de tous les mots du dictionnaire, qui est de l'ordre de $n \times n^5 = n^6$.
Pour ce qui est du coût mémoire, il vaut $SN = N\,n^6$ pour les dictionnaires tabulés et $S = n^6$ pour les dictionnaires binaires.
Dans le cas courant, $N \approx 26 \approx 2n$.
IV.11.b Le test d'appartenance ou d'ajout d'un mot de longueur ℓ a une complexité en temps ℓ dans le cas d'un dictionnaire tabulé, puisqu'on choisit en temps constant la branche où descendre (c'est l'intérêt des tableaux) ; dans le cas d'un dictionnaire binaire, on pourra parcourir l'ensemble des frères d'un nœud, c'est-à-dire décrire tout ou partie d'une frange droite de l'arbre avant de pouvoir descendre sur la gauche : le coût en temps est donc ici en moyenne $\ell N/2$.
On constate que la structure tabulée est plus gourmande en mémoire mais plus efficace en temps, dans le rapport de 1 à N environ à chaque fois.

Question IV.12
On propose le programme 9, page 5.
On a choisi d'écrire une fonction `filsNoeudT : dictTab vect -> (int * dictTab) list` qui à une table t de dictionnaires associe la liste associative contenant les couples (i, d_i) où i est le code d'un caractère et d_i le fils correspondant, limitée aux seuls couples pour lesquels d_i est non vide.

Question IV.13
On propose le programme 10, page 5.
Il faut faire très attention au cas des feuilles, qui ont une représentation tout à fait particulière dans la structure de dictionnaire tabulée (lignes 97–100).

```
let filsNoeudT n =
  let rec boucle i res =
    if i > N then res
    else if n.(i)<>VideT then boucle (i+1) ((i,n.(i)) :: res)
    else boucle (i+1) res
  in
    boucle 0 [] ;;

let rec tabVersBin = function
  | VideT -> VideB
  | NoeudT n ->
      let rec ajouteFils = function
       | [] -> VideB
       | (t,dt) :: q -> NoeudB(tabVersBin dt, t, ajouteFils q)
      in
       ajouteFils (filsNoeudT n) ;;
```

Programme 9 la fonction `tabVersBin`

```
let rec binVersTab = function
  | VideB -> VideT
  | NoeudB(g,x,d) when x < N ->
      ( let fg = binVersTab g in
         match binVersTab d with
           | VideT -> let n = make_vect (N+1) VideT in n.(x) <- fg ; NoeudT n
           | NoeudT n -> n.(x) <- fg ; NoeudT n )
  | NoeudB(g,_,d) -> (* when x=N *)
      ( match binVersTab d with
         | VideT -> let n = make_vect (N+1) VideT in n.(N) <- noeudVide () ; NoeudT n
         | NoeudT n -> n.(N) <- noeudVide () ; NoeudT n ) ;;
```

Programme 10 la fonction `binVersTab`

Partie V — Le mot le plus long

On a choisi dans la suite la structure de dictionnaire tabulé.

Question V.14
On écrit une fonction `extrait : int -> mot -> mot` telle que l'appel `extrait a m` renvoie le mot `m` privé de la première occurrence de la lettre codée `a`, et déclenche l'exception `Not_found` si elle n'y figure pas.
La fonction `listeBranches : dictTab vect -> int list` renvoie la liste des codes de lettres pour lesquelles l'élément de la table n'est pas égal à `VideT` : ce sont les lettres à tester utilement.
La fonction `imprimerMotsDans` s'écrit alors à l'aide d'une fonction récursive `imprimePrefixe` qui reçoit 3 argument : le préfixe (lu de droite à gauche) qui correspond au chemin parcouru dans l'arbre, le mot qui reste à construire, et le nœud courant de l'arbre parcouru.
Il suffit alors de descendre dans l'arbre suivant les branches utiles, et d'effectuer effectivement l'impression au moment où la branche `$` n'est pas vide, c'est-à-dire qu'on passe par une feuille : c'est la ligne 121.

```
101  let rec extrait a = function
102    | [] -> raise Not_found
103    | t :: q -> if t = a then q else t :: (extrait a q) ;;

104  let listeBranches v =
105    let rec boucle branches = function
106      | i when i < N ->
107         if v.(i) <> VideT then boucle (i :: branches) (i+1)
108         else boucle branches (i+1)
109      | _ -> branches
110    in
111      boucle [] 0 ;;

112  let imprimerMotsDans d m =
113    let rec imprimePrefixe prefixeDG m = function
114      | VideT -> ()
115      | NoeudT n ->
116         do_list
117           (function a ->
118              try let m' = extrait a m in imprimePrefixe (a :: prefixeDG) m' n.(a)
119              with Not_found -> ())
120           (listeBranches n) ;
121         if n.(N) <> VideT then imprimer (inverseDe prefixeDG)
122    in
123      imprimePrefixe [] m d ;;
```

Programme 11 la fonction `imprimerMotsDans`

La recherche d'un mot de longueur ℓ permet au passage de trouver tous ses préfixes. Or il y a au plus $\ell!$ sous-mots de longueur ℓ ; chaque recherche ayant un coût ℓ, on peut estimer la complexité de la fonction écrite à $\ell\,\ell!$.

Partie VI — Anagrammes

Question VI.15

On commence par quelques programmes auxiliaires : `chercheMotsDans` effectue la même recherche que `imprimeMotsDans`, mais, au lieu d'imprimer les mots trouver, en renvoie la liste.
La fonction `tri` réalise un tri fusion : en effet on écrit ensuite la fonction `effaceDe : mot -> mot -> mot` qui retire du premier argument les lettres du deuxième. Si une lettre figure dans le deuxième mot mais pas dans le premier, ou bien si elle y figure plus souvent, l'exception `Not_found` est déclenchée. Cette fonction exige que ses arguments soient triés, afin d'être plus efficace.

```
124 let chercheMotsDans d m =
125   let rec cherche liste prefixeDG m = function
126     | VideT -> liste
127     | NoeudT n ->
128        it_list
129          (fun res a ->
130             try let m' = extrait a m in cherche res (a :: prefixeDG) m' n.(a)
131             with Not_found -> res)
132          (if n.(N) <> VideT then (inverseDe prefixeDG) :: liste else liste)
133          (listeBranches n)
134   in
135     cherche [] [] m d ;;

136 let rec tri = function
137   | [] -> []
138   | [a] -> [a]
139   | m -> let m1,m2 = découpe m in fusion (tri m1) (tri m2)
140 and découpe = function
141   | [] -> ([],[])
142   | [a] -> ([a],[])
143   | a::b::q -> let (q1,q2) = découpe q in (a::q1, b::q2)
144 and fusion m1 m2 = match (m1,m2) with
145   | [],_ -> m2
146   | _,[] -> m1
147   | a1::q1, a2::q2 -> if a1 < a2 then a1 :: (fusion q1 m2) else a2 :: (fusion m1 q2) ;;

148 let rec effaceDe m1 m2 =
149   match (m1,m2) with
150     | _,[] -> m1
151     | [],_ -> raise Not_found
152     | a1 :: q1, a2 :: q2 ->
153        if a1 < a2 then a1 :: (effaceDe q1 m2)
154        else if a1 = a2 then effaceDe q1 q2
155        else raise Not_found ;;
```

Programme 12 quelques fonctions auxiliaires

La fonction `anagrammes` retourne la liste des anagrammes constructibles à partir d'un *lexique*, c'est-à-dire de la liste des mots du dictionnaire qui sont sous-mots du mot considéré (c'est ce que renvoie un appel à `chercheMotsDans`).
Elle renvoie un objet du type `mot list list` : chaque élément de la liste résultat est une liste de mots du dictionnaire, ce que nous appellerons une *phrase*, dont la concaténation vaut un anagramme du mot proposé. La fonction `ecrasePhrase` prend une phrase et renvoie le mot constitué des mots de la phrase séparés par des espaces. On en déduit aussitôt la fonction `imprimerAnagrammes`.
Toute la difficulté est donc dans l'écriture de la fonction `anagrammes`. On a utilisé une fonction récursive `combine` qui prend 4 arguments : la liste des phrases-anagrammes déjà trouvées, qu'on essaie d'enrichir, la nouvelle phrase en cours de construction, qu'on devra abandonner éventuellement, les mots disponibles du lexique, et ce qu'il reste à construire du mot de départ pour compléter la phrase en cours.
Si le mot à construire est vide, c'est que la phrase en cours est une nouvelle solution, c'est-à-dire une phrase-anagramme, qu'on ajoute aux solutions déjà trouvées.
Sinon, ou bien il n'y a pas de mot disponible dans le lexique, et on abandonne, en renvoyant les solutions déjà trouvées.
Ou bien on choisit un mot `essai` disponible du lexique, qu'on trie, et qu'on efface du mot `w` à construire, obtenant un nouveau mot `w'`. On cherche alors d'une part les phrases contenant `essai` et qui sont obtenues à partir du lexique entier pour construire `w'`, et d'autre part les phrases n'utilisant pas `essai` pour construire le mot de départ `w`.

```
let anagrammes lexique m =
  let rec combine phrasesTrouvées phraseEnCours disponibles = function
    | [] -> phraseEnCours :: phrasesTrouvées
    | w -> match disponibles with
       | [] -> phrasesTrouvées
       | essai :: reste -> let essai' = tri essai in
         try
           let w' = effaceDe w essai' in
             combine
               (combine phrasesTrouvées (essai :: phraseEnCours) disponibles w')
               phraseEnCours reste w
         with Not_found -> combine phrasesTrouvées phraseEnCours reste w
  in
    combine [] [] lexique m ;;

let rec ecrasePhrase = function
  | [] -> []
  | [ m ] -> m
  | m :: q -> m @ (0 :: (ecrasePhrase q)) ;;

let imprimerAnagrammes dt m =
  let m = tri m in
  let phrases = anagrammes (chercheMotsDans dt m) m in
    do_list (function phrase -> imprimer (ecrasePhrase phrase)) phrases ;;
```

Programme 13 la fonction `imprimerAnagrammes`

Il est bien difficile de donner une valeur intéressante à la complexité de cette fonction : si le lexique est vide, la complexité se réduit au calcul de ce lexique, qui est de l'ordre de $\ell\,\ell!$. Sinon, tout dépend du nombre de mots du lexique, de leurs longueurs ... et des phrases-anagrammes !
La discussion est laissée au lecteur courageux.