CE DUC Y PARLE

1 Structure de trie

Considérons l'ensemble de mots suivant :

```
let mots = ["diane"; "dire"; "diva"; "divan"; "divin"; "do"; "dodo";
            "dodu"; "don"; "donc"; "dont"; "ame"; "ames"; "amen"]
```

On peut représenter cet ensemble sous forme d'un arbre d'arité variable, où un nœud grisé signifie que le mot correspondant (c'est-à-dire le mot qu'on lit en allant de la racine au nœud) appartient au dictionnaire : on parle de *trie* pour cette structure de données.

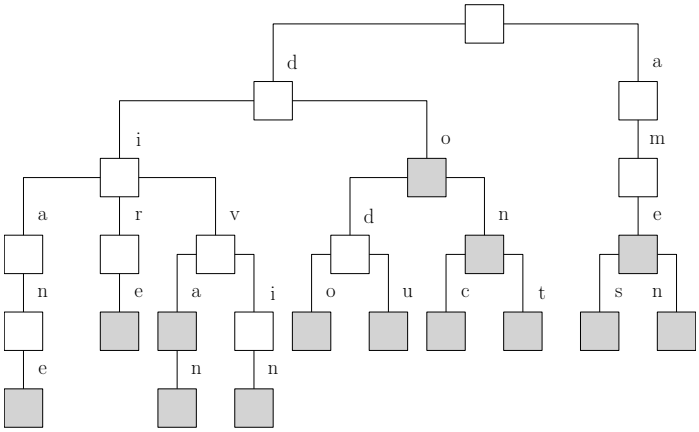

FIGURE XXIII.1 – Représentation arborescente de `mots`.

Pour simplifier la programmation, nous allons utiliser une technique assez courante :

- on choisit un caractère qui n'apparaît dans aucun mot de notre dictionnaire (pour nous, ce sera `$`) ;
- ce caractère devient un « marqueur de fin de mot » : il est ajouté à la fin de tous les mots.

Il n'est alors plus nécessaire de distinguer les nœuds correspondant à un mot du dictionnaire et les autres : les mots du dictionnaire sont exactement ceux qui correspondent aux feuilles de notre arbre.

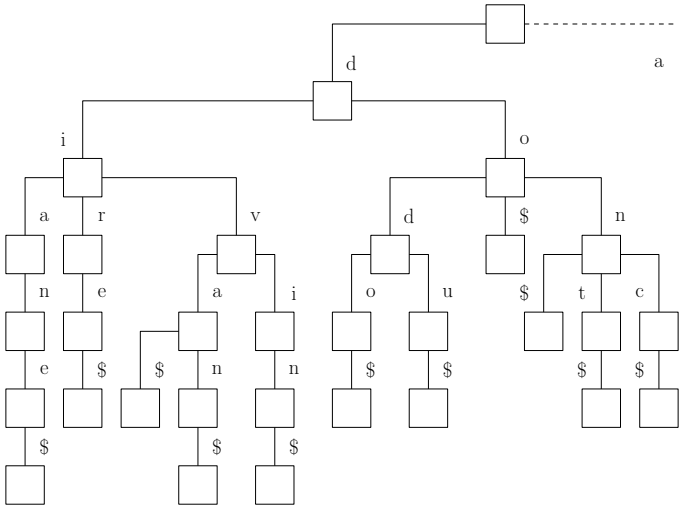

FIGURE XXIII.2 – Partie gauche de l'arbre en figure XXIII.1, avec marqueurs de fin de mot.

Ensuite, nous avons déjà parlé de plusieurs représentations mémoire possibles pour un arbre d'arité quelconque :

- chaque nœud peut contenir un tableau d'enfants (ici, l'enfant correspondant au caractère `c` pourrait être placé dans la case `int_of_char c` du tableau) ;
- chaque nœud peut aussi contenir une liste d'enfants, ou plutôt une liste de couples (caractère, enfant) (puisque les arêtes portent des étiquettes) ;
- enfin, il est possible de « binariser » l'arbre.

C'est cette solution que nous allons choisir, et l'on définit donc le type suivant :

```
type dict =
  | V
  | N of char * dict * dict
```

On obtient alors (en omettant les nœuds `V`) :

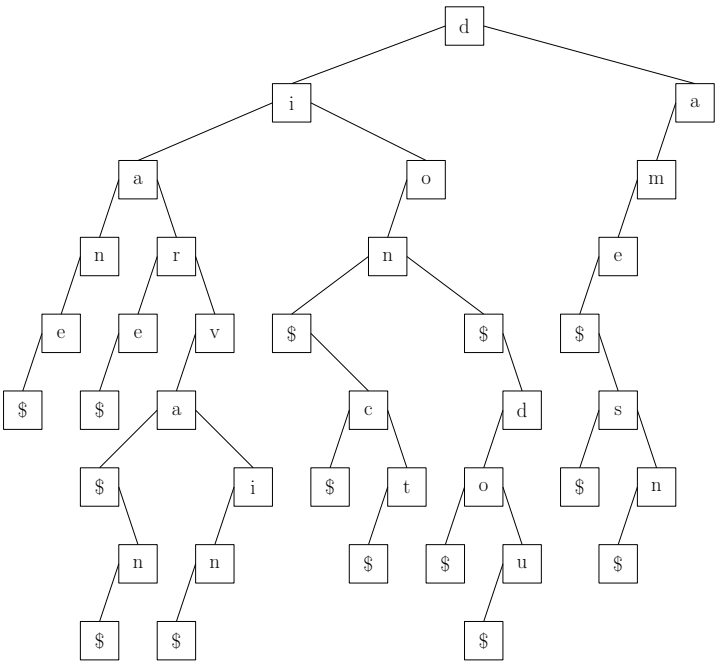

Figure XXIII.3 – Version binaire de l'arbre de la figure XXIII.2.

On impose deux contraintes sur nos arbres :

- un nœud étiqueté par `$` a forcément `V` comme fils gauche ;
- un nœud étiqueté par un caractère autre que `$` n'a jamais `V` comme fils gauche.

Exercice XXIII.1 p. 7

Définir une fonction `est_bien_forme` qui vérifie si un `dict` est bien formé.

```
est_bien_forme : dict -> bool
```

Remarque

À partir de maintenant, les tries passés en argument seront systématiquement supposés bien formés, et toute fonction renvoyant un trie devra obligatoirement renvoyer un trie bien formé.

Un mot (une suite finie de caractères) sera dit *bien formé* s'il contient exactement un caractère `$`, placé en dernière position.

Remarque

Un mot bien formé est donc nécessairement non vide.

Exercice XXIII.2 p. 7

Donner une définition (inductive) de la fonction φ qui à un arbre binaire (bien formé) du type ci-dessus associe un ensemble de mots bien formés. Dans l'exemple ci-dessus, on a :

$$\varphi(t) = \{\texttt{"diane\$"}, \texttt{"dire\$"}, \texttt{"diva\$"}, \texttt{"divan\$"}, \texttt{"divin\$"}, \texttt{"don"}\ \texttt{"donc\$"}, \texttt{"dont\$"}, \texttt{"dodo\$"}, \texttt{"dodu\$"}, \texttt{"ame\$"}, \texttt{"ames\$"}, \texttt{"amen\$"}\}$$

2 Fonctions utilitaires

On définit l'alias de type suivant :

```
type mot = char list
```

Exercice XXIII.3 p. 7

1. Écrire une fonction `mot_of_string` prenant en entrée une chaîne de caractères et renvoyant la liste de ses caractères, avec un caractère `$` rajouté à la fin.

```
mot_of_string : string -> mot
```

```
# mot_of_string "bonjour";;
- : char list = ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$']
```

2. Écrire une fonction `afficher` qui prend en entrée une liste de caractères et affiche le mot correspondant, suivi d'un retour à la ligne. Les éventuels caractères `$` seront ignorés.

```
afficher : mot -> unit
```

```
# afficher_liste ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'; '$'];;
bonjour
- : unit = ()
```

Un objet de type mot sera dit bien formé s'il contient exactement un caractère `$`, placé en dernière position.

3 Opérations élémentaires sur les tries

Exercice XXIII.4 p. 7

1. Écrire une fonction `cardinal` renvoyant le nombre de mots bien formés contenus dans un dictionnaire.

```
cardinal : dict -> int
```

2. Écrire une fonction `appartient` qui détermine si un certain mot bien formé appartient à un dictionnaire.

```
appartient : dict -> mot -> bool
```

Exercice XXIII.5 p. 8

1. Écrire une fonction `ajouter` qui ajoute un mot, supposé bien formé, à un dictionnaire.
2. Écrire une fonction `dict_of_list` prenant en entrée une liste de chaînes de caractères et renvoyant un dictionnaire contenant exactement les mots de cette liste (auxquels on a ajouté un `$`).

```
ajouter : dict -> mot -> dict
dict_of_list : string list -> dict
```

Exercice XXIII.6 p. 8

1. Écrire une fonction `afficher_mots` qui affiche tous les mots bien formés appartenant à un dictionnaire (sans les `$` finals), à raison d'un mot par ligne.
2. Écrire une fonction `longueur_maximale` qui renvoie la longueur maximale d'un mot bien formé du dictionnaire (on ne comptera pas le `$` final, et l'on renverra -1 s'il n'y a pas de mot bien formé).
3. Écrire une fonction `afficher_mots_longs` qui affiche tous les mots de longueur supérieure ou égale à l'entier passé en argument (toujours sans compter le `$`).

```
afficher_mots : dict -> unit
longueur_maximale : dict -> int
afficher_mots_longs : dict -> int -> unit
```

4 Lecture de fichier

Vous trouverez sur le serveur quelques fichiers contenant des mots, dont le format est très simple : chaque mot est sur une ligne. Ces mots ne contiennent que des lettres minuscules de a à z, sans signe diacritique (autrement dit, les accents, cédilles et autres ont été retirés des fichiers en français).

- Le fichier `ab.txt` contient 9 940 mots anglais, commençant tous par a ou par b.
- Le fichier `10000.txt` contient les 10 000 mots anglais les plus courants.
- Le fichier `nettoye.txt` contient 336 531 mots français, débarassés de leurs signes diacritiques.

Dans tous les cas, il y a quelques doublons (dûs à la suppression des signes diacritiques).

Exercice XXIII.7 p. 9

Écrire une fonction `lire_fichier` qui prend en entrée un nom de fichier (ou plutôt un chemin relatif vers un fichier) et renvoie le dictionnaire correspondant. On supposera que le format est celui décrit ci-dessus (un mot par ligne).

```
lire_fichier : string -> dict
```

```
# cardinal (lire_fichier "ab.txt");;
- : int = 9938
# cardinal (lire_fichier "10000.txt");;
- : int = 9989
# cardinal (lire_fichier "nettoye.txt");;
- : int = 323422
```

5 Filtrage

Exercice XXIII.8 p. 9

1. Écrire une fonction `calculer_occurrences` qui prend en entrée une chaîne de caractères `s` et renvoie un `int array` de longueur 256 tel que `t.(i)` soit égal au nombre d'occurrences de `int_of_char i` dans `s`.
2. Écrire une fonction `afficher_mots_contenus` qui prend en entrée un mot sous forme de chaîne de caractères (sans `$` final) et un dictionnaire, et affiche tous les mots du dictionnaire que l'on peut former en utilisant tout ou partie des lettres du mot fourni (en tenant compte des répétitions).
3. Écrire une fonction `afficher_anagrammes` qui prend en entrée un mot sous forme de chaîne de caractères et affiche toutes ses anagrammes présentes dans le dictionnaire. Une *anagramme* d'un mot est un mot constitué exactement des mêmes lettres (avec le même nombre d'occurrences) mais dans un ordre différent (on considérera qu'un mot est anagramme de lui-même).
4. Quel est sont les mots français les plus courts contenant toutes les voyelles ?

```
calculer_occurrences : string -> int array
afficher_mots_contenus : dict -> string -> unit
afficher_anagrammes : dict -> string -> unit
```

Exercice XXIII.9 p. 11

1. Écrire une fonction `filtrer_mots_contenus` qui prend les mêmes arguments que `afficher_mots_contenus` mais renvoie un dictionnaire contenant les mots que l'on peut former. On produira directement le dictionnaire, sans commencer par produire la liste des mots.
2. Écrire une fonction `filtrer_mots_contenant` qui fait la même chose que la précédente, sauf qu'on s'intéresse cette fois aux mots *à partir desquels* on peut former le mot fourni (c'est-à-dire ceux contenant toutes les lettres du mot fourni, en tenant compte des répétitions).
3. Écrire une fonction similaire `filtrer_anagrammes`.

6 Décomposition en anagrammes

On appelle *décomposition en anagrammes* d'un mot m dans un dictionnaire d une suite de mots de d qui, mis bout à bout, forment un anagramme de m. Par exemple, `"sans ame"`, `"sa mes na"`, `"a mes ans"` sont des décompositions possibles de `"massena"` avec un dictionnaire français standard.

Exercice XXIII.10 p. 13

Dans cet exercice, on s'autorise à générer plusieurs fois une décomposition : par exemple, `"sans ame"` et `"ame sans"` (autrement dit, une décomposition en n mots sera générée $n!$ fois).

1. Écrire une fonction `afficher_decompositions` qui affiche toutes les décompositions en anagrammes d'un mot dans un dictionnaire.
2. Écrire une fonction `decompositions` qui renvoie un dictionnaire contenant toutes ces décompositions. La décomposition `"sans ame"`, par exemple, sera stockée comme un mot de huit caractères (sans compter le `$` final), avec un caractère « espace » entre le « s » et le « a ».

```
afficher_decompositions : dict -> string -> unit
decompositions : dict -> string -> dict
```

Exercice XXIII.11 p. 14

Écrire une fonction `decompositions_uniques` qui génère le dictionnaire des décompositions dans lequel deux décompositions ne différant que par l'ordre des mots sont considérées comme identiques (et ne contenant donc que l'une de ces décompositions). On pourra par exemple ne générer que les décompositions pour lesquelles la suite des mots est croissante (dans l'ordre lexicographique).

```
decompositions_uniques : dict -> string -> dict
```

Solutions

Correction de l'exercice XXIII.1 page 2

```
let rec est_bien_forme = function
  | V -> true
  | N ('$', V, d) -> est_bien_forme d
  | N (_, V, _) | N ('$', _, _) -> false
  | N (c, g, d) -> est_bien_forme g && est_bien_forme d
```

Correction de l'exercice XXIII.2 page 3

En notant cs le mot formé du caractère c suivi du mot s, on a :

- $\varphi(V) = \emptyset$
- $\varphi(N(\$, V, d)) = \{\$\} \cup \varphi(d)$
- $\varphi(N(c, g, d)) = \{cs \mid s \in \varphi(g)\} \cup \varphi(d)$ si $c \neq \$$.

Correction de l'exercice XXIII.3 page 3

```
let mot_of_string s =
  let n = String.length s in
  let rec aux i =
    if i = n then ['$']
    else s.[i] :: aux (i + 1) in
  aux 0

let rec afficher = function
  | [] -> print_newline ()
  | '$' :: xs -> afficher xs
  | x :: xs -> print_char x; afficher xs
```

Correction de l'exercice XXIII.4 page 3

Le cardinal est égal au nombre de nœuds de la forme `N ('$', V, d)`.

```
let rec cardinal = function
  | V -> 0
  | N ('$', V, d) -> 1 + cardinal d
  | N (c, g, d) -> cardinal g + cardinal d
```

Pour `appartient`, il n'y a pas de raison de traiter les nœuds étiquetés `'$'` séparément :

```
let rec appartient dict mot =
  match dict, mot with
  | V, [] -> true
  | N (c, g, d), x :: xs ->
    if c = x then appartient g xs else appartient d mot
  | _ -> false
```

Correction de l'exercice XXIII.5 page 4

1. La structure est similaire à celle de `appartient`.

```
let rec ajouter dict mot =
    match dict, mot with
    | V, [] -> V
    | V, x :: xs -> N (x, ajouter V xs, V)
    | N (c, g, d), x :: xs ->
      if c = x then N (c, ajouter g xs, d) else N (c, g, ajouter d mot)
    | _ -> failwith "mal formé"
```

2. La solution la plus simple à écrire :

```
let rec dict_of_list u =
  match u with
  | [] -> V
  | s :: tl -> ajouter (dict_of_list tl) (mot_of_string s)
```

Notons que cette fonction n'est pas récursive terminale, ce qui peut être gênant si jamais on a une très longue liste de mots à ajouter (mais ce ne sera pas le cas ici). On peut donc préférer écrire :

```
let dict_of_list u =
  let rec aux restant d =
    match restant with
    | [] -> d
    | s :: tl -> aux tl (ajouter d (mot_of_string s)) in
  aux u V
```

Ces deux fonctions ne font pas les ajouts dans le même ordre, et ne renvoient pas, en général, le même dictionnaire. Cependant, elles sont toutes les deux correctes : dans les deux cas, le dictionnaire est bien formé et contient exactement les mots de la liste.

Correction de l'exercice XXIII.6 page 4

1. On parcourt l'arbre en maintenant à jour une liste `prefixe` contenant le mot lu (actuellement) depuis la racine. Quand on tombe sur un nœùd `N ('$', V, d)`, on affiche le mot (en retournant la liste, puisqu'elle se construit naturellement « à l'envers »). Pour un nœud `N (c, g, d)` avec `c` un caractère autre que `$`, on descend à gauche en ajoutant `c` au préfixe, et à droite sans modifier le préfixe.

```
let afficher_mots dict =
  let rec aux dict prefixe =
    match dict with
    | V -> ()
    | N ('$', V, d) -> afficher (List.rev prefixe); aux d prefixe
    | N (c, g, d) -> aux g (c :: prefixe); aux d prefixe in
  aux dict []
```

2. Il n'est pas nécessaire de traiter séparément les nœuds étiquetés `$` :

```
let rec longueur_maximale = function
  | V -> -1
  | N (c, g, d) -> max (1 + longueur_maximale g) (longueur_maximale d)
```

3. On procède comme pour `afficher_mots`, en ajoutant un paramètre donnant la longueur actuelle du préfixe. On pourrait se contenter de la calculer avant de décider si on affiche le mot sans que cela n'impacte réellement la complexité.

```
let afficher_mots_longs dict n =
  let rec aux dict prefixe i =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      if i >= n then afficher (List.rev prefixe);
      aux d prefixe i
    | N (c, g, d) ->
      aux g (c :: prefixe) (i + 1);
      aux d prefixe i in
  aux dict [] 0
```

Correction de l'exercice XXIII.7 page 4

Le plus simple est sans doute d'utiliser une boucle `while true` dont on sortira *via* une exception :

```
let lire_fichier f =
  let ic = open_in f in
  let dict = ref V in
  try
    while true do
      let s = input_line ic in
      dict := ajouter !dict (mot_of_string s)
    done;
    V
  with
  | End_of_file -> !dict
```

Correction de l'exercice XXIII.8 page 5

1. Aucune difficulté, on utilise la fonction `String.iter` (similaire à `Array.iter` et `List.iter`) mais l'on pourrait bien sûr faire une boucle `for` :

```
let calculer_occurrences s =
  let occs = Array.make 256 0 in
  for i = 0 to String.length s - 1 do
    let x = int_
    occs.(int_of_char c) <- occs.(int_of_char c) + 1
  done;
  occs
```

2. La version la plus simple à comprendre est sans doute celle-ci :

```
let afficher_mots_contenus dict s =
  let rec aux dict prefixe occs =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      afficher (List.rev prefixe);
      aux d prefixe occs
    | N (c, g, d) ->
      aux d prefixe occs;
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        let nv_occs = Array.copy occs in
        nv_occs.(i) <- nv_occs.(i) - 1;
        aux g (c :: prefixe) nv_occs
      end in
  aux dict [] (calculer_occurrences s)
```

Il faut bien faire une copie de `occs` et modifier cette copie (et non l'original) : sinon, lorsqu'on remonte dans l'arbre après avoir terminé l'exploration d'une branche, les lettres « consommées » ne sont pas rendues.

On peut cependant faire plus efficace en utilisant un seul tableau `occs` pour tous les appels récursifs. Il faut alors penser à le « remettre en état » quand l'appel sur la branche de gauche se termine :

```
let afficher_mots_contenus_bis dict s =
  let occs = calculer_occurrences s in
  let rec aux dict prefixe =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      afficher (List.rev prefixe);
      aux d prefixe
    | N (c, g, d) ->
      aux d prefixe;
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        occs.(i) <- occs.(i) - 1;
        aux g (c :: prefixe);
        occs.(i) <- occs.(i) + 1
      end in
  aux dict []
```

Quasiment toutes les fonctions qu'il nous reste à écrire sont des variations sur cette idée : dans le corrigé, on utilisera la deuxième variante, un peu plus concise et plus efficace, mais la première variante conviendrait aussi.

3. La fonction est extrêmente similaire, seul le test pour afficher diffère :

```
let est_nul t =
  let n = Array.length t in
  let i = ref 0 in
  while !i < n && t.(!i) = 0 do
    incr i
  done;
  !i = n
```

```
let afficher_anagrammes dict s =
  let occs = calculer_occurrences s in
  let rec aux dict prefixe =
    match dict with
    | V -> ()
    | N ('$', V, d) ->
      if est_nul occs then afficher (List.rev prefixe);
      aux d prefixe
    | N (c, g, d) ->
      aux d prefixe;
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        occs.(i) <- occs.(i) - 1;
        aux g (c :: prefixe);
        occs.(i) <- occs.(i) + 1
      end in
  aux dict []
```

Correction de l'exercice XXIII.9 page 5

1. Ce n'est pas très différent de l'affichage. Attention cependant à renvoyer un arbre bien formé : c'est le rôle du `if g' = V then d' else N (c, g', d')`.

```
let rec filtrer_contenus_occs dict occs =
  match dict with
  | V -> V
  | N ('$', V, d) ->
    N ('$', V, filtrer_contenus_occs d occs)
  | N (c, g, d) when occs.(int_of_char c) = 0 ->
    filtrer_contenus_occs d occs
  | N (c, g, d) ->
    let i = int_of_char c in
    let d' = filtrer_contenus_occs d occs in
    occs.(i) <- occs.(i) - 1;
    let g' = filtrer_contenus_occs g occs in
    occs.(i) <- occs.(i) + 1;
    if g' = V then d'
    else N (c, g', d')

let filtrer_mots_contenus dict s =
  let occs = calculer_occurrences s in
  filtrer_contenus_occs dict occs
```

2. Très similaire :

```
let est_negatif t =
  let n = Array.length t in
  let rec loop i =
    i = n || t.(i) <= 0 && loop (i + 1) in
  loop 0
```

```
let filtrer_contenant_occs dict s =
  let occs = calculer_occurrences s in
  let rec aux = function
    | V -> V
    | N (c, g, d) when est_negatif occs -> N (c, g, d)
    | N (c, g, d) ->
      let i = int_of_char c in
      let d' = aux d in
      occs.(i) <- occs.(i) - 1;
      let g' = aux g in
      occs.(i) <- occs.(i) + 1;
      if g' = V then d'
      else N (c, g', d') in
  aux dict
```

3. Toujours le même principe :

```
let filtrer_anagrammes_occs dict occs =
  let rec aux = function
    | V -> V
    | N ('$', V, d) ->
      if est_nul occs then N ('$', V, V)
      else aux d
    | N (c, g, d) when occs.(int_of_char c) = 0 -> aux d
    | N (c, g, d) ->
      let i = int_of_char c in
      let d' = aux d in
      occs.(i) <- occs.(i) - 1;
      let g' = aux g in
      occs.(i) <- occs.(i) + 1;
      if g' = V then d' else N (c, g', d') in
  aux dict

let filtrer_anagrammes dict s =
  let occs = calculer_occurrences s in
  filtrer_anagrammes_occs dict occs
```

4. On écrit une fonction pour afficher les mots les plus courts d'un dictionnaire :

```
let afficher_mots_les_plus_courts dict =
  let rec aux prefixe d =
    match d with
    | V -> [], max_int
    | N ('$', V, d) -> [List.rev prefixe], 0
    | N (c, g, d) ->
      let mots_g, l_g = aux (c :: prefixe) g in
      let mots_d, l_d = aux prefixe d in
      if l_g >= l_d then mots_d, l_d
      else if l_g = l_d - 1 then mots_g @ mots_d, l_d
      else mots_g, l_g + 1 in
  let mots, _ = aux [] dict in
  List.iter afficher mots
```

On peut ensuite l'appliquer au dictionnaire constitué des mots contenant toutes les voyelles, et obtenir : *boyaudier, guerroyai, noyautiez, paumoyiez, rougeoyai*.

Correction de l'exercice XXIII.10 page 5

1. La différence fondamentale, c'est qu'il y a deux cas quand on arrive sur un nœud `N ('$', V, d)` :
   - soit on a utilisé toutes les lettres, et dans ce cas on a fini de décomposer ;
   - soit il nous reste des lettres, et dans ce cas il faut ajouter au moins un mot à la décomposition, en repartant au sommet de l'arbre sans changer le tableau `occs`.

```
let afficher_decompositions dict mot =
  let occs = calculer_occurrences mot in
  let rec aux prefixe = function
    | V -> ()
    | N ('$', V, d) ->
      if est_nul occs then afficher (List.rev prefixe)
      else begin
        aux (' ' :: prefixe) dict;
        aux prefixe d
      end
    | N (c, g, d) ->
      let i = int_of_char c in
      if occs.(i) > 0 then begin
        occs.(i) <- occs.(i) - 1;
        aux (c :: prefixe) g;
        occs.(i) <- occs.(i) + 1
      end;
      aux prefixe d in
  aux [] dict
```

2. Pas de problème majeur si l'on a compris la question précédente. On choisit de restreindre le dictionnaire aux mots que l'on peut former à partir de `mot`, ce qui rend la fonction légèrement plus efficace. Notons quand même que, pour que l'arbre renvoyé soit bien formé, la distinction de cas ligne 10 est nécessaire.

```
1  let decompose_anagrammes dict mot =
2    let occs = calculer_occurrences mot in
3    let initial = filtrer_mots_contenus dict mot in
4    let rec aux = function
5      | V -> V
6      | N ('$', V, d) ->
7        if est_nul occs then N ('$', V, V)
8        else
9          let g' = aux initial in
10         if g' <> V then N (' ', g', aux d) else aux d
11     | N (c, _, d) when occs.(int_of_char c) = 0 -> aux d
12     | N (c, g, d) ->
13       let i = int_of_char c in
14       let d' = aux d in
15       occs.(i) <- occs.(i) - 1;
16       let g' = aux g in
17       occs.(i) <- occs.(i) + 1;
18       if g' = V then d' else N (c, g', d') in
19   aux initial
```

Correction de l'exercice XXIII.II page 6

Une solution possible est de se souvenir du mot précédent de la décomposition, et d'imposer que le mot actuel vienne après ce mot précédent dans l'ordre lexicographique, ce qui n'est pas complètement évident à faire sans rendre le code excessivement lourd. Ici :

- si ce qui reste du mot précédent est vide, c'est bon ;
- si la prochaine lettre du mot précédent est strictement inférieure à la lettre que l'on choisit (en suivant une branche gauche), alors on peut choisir n'importe quoi ensuite (ce qu'on traduit en remplaçant le mot précédent par la liste vide) ;
- si elle est égale à la lettre choisie, on passe à la lettre suivante du mot précédent ;
- si elle est strictement supérieure, on ne peut pas prendre cette branche.

```
let supprimer_tete = function
  | [] -> []
  | _ :: xs -> xs

let comparer_tete (x : char) v =
  match v with
  | y :: _ ->
    if x < y then -1
    else if x = y then 0
    else 1
  | [] -> 1

let decompose_anagrammes_unique dict mot =
  let occs = calculer_occurrences mot in
  let initial = filtrer_contenus_occs dict occs in
  let rec aux precedent actuel dict =
    match dict with
    | V -> V
    | N ('$', V, d) when precedent = [] ->
      if est_nul occs then N ('$', V, V)
      else
        let g' = aux (List.rev actuel) [] initial in
        if g' <> V then N (' ', g', aux precedent actuel d)
        else aux precedent actuel d
    | N (c, _, d) when occs.(int_of_char c) = 0 -> aux precedent actuel d
    | N (c, g, d) ->
      let i = int_of_char c in
      let d' = aux precedent actuel d in
      let comparaison = comparer_tete c precedent in
      if comparaison = -1 then d'
      else begin
        occs.(i) <- occs.(i) - 1;
        let precedent' =
          if comparaison = 0 then supprimer_tete precedent
          else [] in
        let g' = aux precedent' (c :: actuel) g in
        occs.(i) <- occs.(i) + 1;
        if g' = V then d' else N (c, g', d')
      end in
  aux [] [] initial
```