Table des matières

Arbres lexicographiques

solution : **Erreurs fréquentes :** Ne pas comprendre le rôle du mot vide, et d'un nœud terminal à la racine. Dans cette situation, vous êtes tentés de mettre les cas d'arrêts aux mots de une lettre, ce qui est lourd.

On représentera les mots sous Caml non pas par le type `string` mais une liste de caractères.

```
type mot = char list;;
```

Exemple : L'expression `[ 'f'; 'a'; 'c'; 'e' ]` est associée au mot « face », de longueur 4.
N.B. En Caml, un caractère est entouré de guillemets anglaises simples, (touche 4 d'un clavier azerty), tandis qu'une chaîne de caractères est entourées de guillemets anglaises doubles (touche 3).

Les arbres lexicographiques sont des arbres utilisés pour représenter des ensembles de mots. Les arêtes sont étiquetés par un caractère. La suite des caractères qui étiquettent les arêtes le long d'un chemin de la racine de l'arbre jusqu'à un nœud forme donc un mot.

Certains nœuds sont appelés nœuds « terminaux ». Les mots représentés par l'arbre sont les mots obtenus en suivant un chemin de la racine jusqu'à un nœud terminal.

Attention : un nœud terminal n'est pas forcément une feuille.

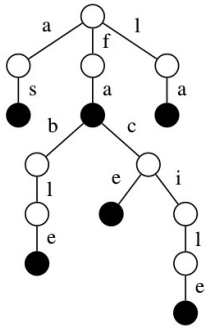

FIGURE 1 – Un exemple d'arbre lexicographique représentant l'ensemble {"as", "fa", "fable", "facile", "la"}. Les nœuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche.

Un arbre lexicographique est représenté par le type `arbre_lex` et le type auxiliaire `fils` ci-dessous :

```
type arbre_lex = Noeud of bool * fils
and fils = (char * arbre_lex) list;;
```

Dans l'appel `Noeud(terminal, fils)`, `terminal` est un drapeau booléen qui indique si le nœud est terminal ou pas. Et `fils` contient la liste des fils ainsi que les étiquettes des arêtes qui y mènent. C'est une liste de couples de la forme `(c, f)` tel que `c` est l'étiquette de l'arête menant au fils `f`.

En outre, un arbre lexicographique sera dit *valide* si pour chaque nœud, les fils sont rangés dans l'ordre alphabétique des étiquettes des arêtes qui y mènent, et si toutes les feuilles sont des nœuds terminaux.

Sauf mention du contraire, on supposera que tous les arbres lexicographiques utilisés sont valides, et dans les question demandant de créer un arbre, on veillera à créer un arbre valide.

Remarque : Ainsi les mots dans un arbre lexicographiques sont-ils rangé dans l'ordre alphabétique. Ceci permettra d'accélérer les recherches. Un tel arbre est donc semble à un arbre binaire de recherche, à la différence près qu'il n'est pas binaire, puisqu'un nœud peut avoir autant de fils qu'il y a de lettres dans l'alphabet choisi.

1. Écrire une fonction `mot_of_string` de type `string -> mot` prenant en entrée une chaîne de caractère, de type `string` et la convertissant en une liste de lettres. On pourra écrire une fonction récursive auxiliaire prenant un argument supplémentaire : la position actuelle dans la chaîne de caractères.
   solution :

```
let rec mot_of_stringAux i m=
  (* Renvoie la liste de caractères [m.[i]; ...; m.[n-1]], où n = String.length m *)
  if i >= String.length m then []
  else m.[i] :: mot_of_stringAux (i+1) m;;

let mot_of_string = mot_of_stringAux 0;;
```

2. L'expression suivante définit un arbre `exemple` :

```
  let feuille= Noeud(true,[]);;
  let exemple=
  Noeud(false,[
      'f',Noeud(false,[
          'a', Noeud(true,[
               ('c', feuille)
                          ]
                    )
      ;
          ('i', feuille)
          ])
  ]);;
```

   Dessiner l'arbre lexicographique correspondant, et donner l'ensemble de mots qu'il représente.

3. Dans quelle situation un nœud terminal qui n'est pas une feuille est-il utile ?
   solution : Lorsqu'un mot est préfixe d'un autre.

4. Dans quelle situation la racine est-elle un nœud terminal ?
   solution : Lorsque le mot vide est dans l'ensemble de mots à représenter. **N.B.** Cette situation sera utile lors des appels récursifs des fonctions à venir.

5. Écrire en CaML une fonction `arbre_of_mot` de type `mot -> arbre_lex` telle que l'appel `(arbre_of_mot m)` sur un mot `m` renvoie un arbre lexicographique contenant uniquement le mot `m`.
   solution :

```
let rec arbre_of_mot = function
  |[]-> feuille
  |t::q -> Noeud(false, [(t, arbre_of_mot q)])
;;
```

6. Écrire en CaML une fonction `nb_mots` de type `arbre_lex -> int` telle que l'appel `(nb_mots a)` sur un arbre lexicographique `a` renvoie le nombre de mots contenus dans l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
   solution : Il s'agit tout simplement de compter le nombre de nœuds terminaux.

```
let rec nb_mots (Noeud(term, fils)) = (* Comme il 'ny a 'quun seul constructeur pour le
    ↪ type arbre, je peux utiliser le raccourci ci-contre. *)
    if term then 1 + nb_mots_foret fils
    else nb_mots_foret fils

and nb_mots_foret = function
    |[]   -> 0
    |(_,a)::q  -> nb_mots a + nb_mots_foret q
;;
```

7. Écrire également une fonction calculant le nombre de feuilles d'un arbre lexicographique.
   solution :

```
''
let rec nb_feuilles  a =
  match a with
    | Noeud(_,[])  -> 1
    | Noeud(_,fils)-> nb_feuilles_foret fils

and nb_feuilles_foret lf =
```

```
    match lf with
    |[]   -> 0
    |(_,a)::q  -> nb_feuilles a + nb_feuilles_foret q
;;
```

8. Écrire une fonction `prefixed` prenant un caractère x et une liste de mots l et remvoyant la liste obtenue en ajoutant x devant chaque mot de l. Bonus à qui l'écrit en une ligne.
solution :

```
''
let prefixed x (l:mot list)=
    (* Renvoie la liste contenant les mots de l prefixée 'dun x. *)
    List.map (fun m-> x::m) l;;
```

9. Écrire en CaML une fonction `mots_of_arbre` de type `arbre_lex -> mot list` telle que l'appel `(mots_of_arbre a)` sur un arbre lexicographique `a` renvoie la liste des mots contenus dans `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
solution :

```
'''
let rec mots_of_arbre a=
    (* Renvoie la liste des mots contenus dans a. *)
    match a with
    |Noeud(true, fils)-> [] :: mots_of_arbre_foret fils (* le mot vide est contenu dans
        ↪ a*)
    |Noeud(false, fils)-> mots_of_arbre_foret fils

and mots_of_arbre_foret f=
  match f with
  |[]-> []
  |(lettre, a)::q -> (prefixed lettre (mots_of_arbre a))
                    @ mots_of_arbre_foret q
;;
```

10. Écrire en CaML une fonction `appartient` de type `mot -> arbre_lex -> bool` telle que l'appel `(appartient m a)` sur un mot `m` et un arbre lexicographique valide `a` renvoie la valeur `true` si et seulement si l'arbre `a` contient le mot `m`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives. La complexité de la recherche d'un mot m doit être en $O(|m|)$. On justifiera cette complexité.
solution : Comme remarqué question **??**, ne pas oublié le cas du mot vide. Dans ce cas, renvoyer le booléen qui est à la racine.

```
''
let rec appartient (m:mot) a =
    (* Indique si le mot m est contenu dans l'arbre a. *)
    match m, a with
    |[], Noeud(term,_) -> term (* Le mot vide est reconnu ssi la racine est terminale *)
    |t::q, Noeud(_,fils)-> appartient_foret t q fils

and appartient_foret lettre mot f =
   match f with
   |[] -> false
   |(x,a)::q when x=lettre -> appartient mot a
   |(x,_)::q when x<lettre -> appartient_foret lettre mot q
   |_ -> false
;;
```

11. Écrire une fonction `ajout` de type `mot -> arbreLex -> arbreLex` prenant en entrée un mot `m` et un arbre lexicographique `a` et renvoyant l'arbre obtenu en rajoutant `m` dans `a`.
solution :

```
'''
let rec ajout m a=
    match m, a with
    |[], Noeud(_, fils)-> Noeud(true,fils)
```

```
    |t::q, Noeud(term, fils)  -> Noeud(term, ajout_foret t q fils)

and ajout_foret lettre mot f=
   match f with
   |[] -> [(lettre, arbre_of_mot mot)]
   |(x,a)::q when lettre <x  -> (lettre, arbre_of_mot mot)::f
   |(x,a)::q when lettre=x   -> (x, ajout mot a)::q
   |(x,a)::q                 -> (x,a):: ajout_foret lettre mot q
;;
```

12. En déduire une fonction `arbre_of_list` prenant une liste de mots et renvoyant un arbre lexicographique contenant les mêmes mots.
solution : On peut le faire en une ligne en utilisant un `List.fold` avec l'opération `ajout`.

```
'''
let arbre_of_list (l:mot list)=
    List.fold_right ajout l (Noeud(false,[]))
;;
```

13. Le fichier `mots_francais.txt` contient tous les mots du français, un par ligne. Voici les commandes de base pour manipuler un fichier :
    - Ouvrir un fichier : `let entree = open_in "adresse du fichier"in ...`
      L'identifiant `entree` contient alors un objet de type « `in_channel` ».
    - Lire une ligne : `input_line` de type `in_channel -> string`. Cette fonction lève l'exception `End_of_file` lorsqu'on est au bout du fichier.
      On pourra utiliser la fonction suivante :

```
let lit_une_ligne entree=
  (* Renvoie la prochaine ligne du canal d'entrée fourni privé de son dernier
     ↪ caractère (le retour chariot), ou "" si on est au bout du fichier. *)
  try
    let ligne = input_line entree in
    let n = String.length ligne in String.sub ligne 0 (n-1)
  with
  |End_of_file -> ""
;;
```

      Pour des raisons de performance, puisque le fichier à lire est gros, il faudra écrire une fonction auxiliaire avec accumulateur.

    Écrire une fonction pour charger tous les mots du français dans un arbre lexicographique.
    solution :

```
'''
let arbre_of_txt chemin=
  let entree=open_in chemin in

  let rec lit_fichier accu=
    let ligne = lit_une_ligne entree in
    if ligne = "" then accu
    else lit_fichier (ajout (mot_of_string ligne) accu)
  in
  lit_fichier (Noeud(false, []));;
```

14. En déduire une fonction `est_francais` permettant de tester si un mot est français.
solution :

```
'''
let est_francais m = appartient (mot_of_string m) dico_francais;;
(*-fin -*)
est_francais "Noël";;
 est_francais "élève";;
est_francais "Banach";;
est_francais "ban";;
```

En seconde année, nous verrons comment calculer la « distance » entre deux mots, c'est-à-dire le nombre de fautes de frappe pour passer de l'un à l'autre. En combinant cela avec ce que nous venons de faire, on obtient un correcteur orthographique.