Problème d'informatique : arbres lexicographiques (d'après CCP)

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 simples (en pratique d'apostrophes, touche 4 du clavier), tandis qu'une chaîne de caractères est entourées de guillemets anglais doubles (touche 3 du clavier).

Les arbres lexicographiques sont des arbres utilisés pour représenter des ensembles de mots. Les arêtes sont étiquetés par un caractère. La séquence 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.

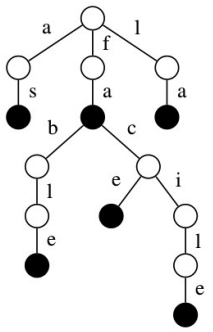

FIGURE 1 – Un exemple d'arbre lexicographique représentant l'ensemble {"as", "fa", "fable", "facile", "la"}. Les nœuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche.

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.

1. Écrire une fonction `mot_of_string` 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.
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 ?
4. Dans quelle situation la racine est-elle un nœud terminal ?
5. Écrire en CaML une fonction `creer` de type `mot -> arbre_lex` telle que l'appel `(creer m)` sur un mot `m` renvoie un arbre lexicographique contenant uniquement le mot `m`.
6. Écrire en CaML une fonction `compter` de type `arbre_lex -> int` telle que l'appel `(compter 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.
7. Écrire également une fonction calculant le nombre de feuilles d'un arbre lexicographique.
8. Écrire en CaML une fonction `accepter` de type `mot -> arbre_lex -> bool` telle que l'appel `(accepter 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.
9. Écrire une fonction `prefixe` prenant un caractère x et une liste de mots l et remvoyant la liste obtenue en ajoutant x devant chaque mot de l.
10. Écrire en CaML une fonction `extraire` de type `arbre_lex -> mot list` telle que l'appel `(extraire 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.
11. Écrire une fonction `ajoute` 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`.
12. En déduire une fonction `abre_of_list` prenant une liste de mots et renvoyant un arbre lexicographique contenant les mêmes mots.
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. Comme il n'y a pas de moyen simple de savoir si on est bout du fichier, on passe en général par cette exception, en écrivant une fonction de la forme :

```
let rec lecture f=
    try
        quelque chose utilisant input_line f et réappelant lecture f pour lire la
    ↪ suite
    with
        |End_of_file -> à renvoyer quand on est au bout du fichier
;;
```

      Si l'expression dans le « try » peut être évaluée sans problème, elle est renvoyée, mais si on rencontre l'exception «`End_of_file`», alors c'est l'autre expression qui est évaluée et renvoyée.

    Écrire une fonction pour charger tous les mots du français dans un arbre lexicographique.
14. En déduire une fonction `est_francais` permettant de tester si un mot est français.

Dans un chapitre ultérieur, 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.