DM - Arbres Lexicographiques

On définit les types :

[1]:
```
type mot = char list;;
type arbre_lex = Noeud of bool*fils
and fils = (char* arbre_lex) list;;
```

[1]:
```
type mot = char list
```

[1]:
```
type arbre_lex = Noeud of bool * fils
and fils = (char * arbre_lex) list
```

1 - On a :

[2]:
```
let rec mot_of_string_aux chaine i=
  if i < String.length chaine then
  (String.get chaine i)::(mot_of_string_aux chaine (i+1))
  else []
;;
let mot_of_string chaine =
(* Convertie une chaine de caractère en liste de char *)
  mot_of_string_aux chaine 0
;;
```

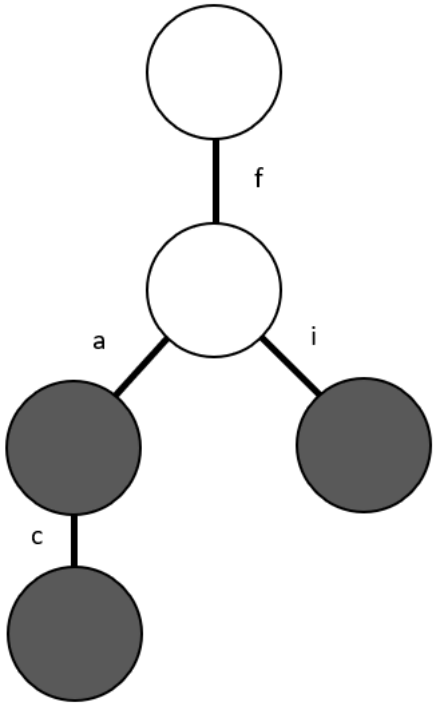
[2]:
```
val mot_of_string_aux : string -> int -> char list = <fun>
```

[2]:
```
val mot_of_string : string -> char list = <fun>
```

[3]:
```
mot_of_string "test";;
```

[3]:
```
- : char list = ['t'; 'e'; 's'; 't']
```

2 - L'ensemble des mots est {fac,fa,fi} et on a l'arbre suivant :



3 - Un noeud Terminal qui n'est pas une feuille est utile pour former plusieurs mots qui commence de la même manière

4 - La racine est un noeud terminal si - L'arbre ne contient que un seul mot - Ce seul mot a un seul caractère

5 - On a :

[4]:
```
let rec creer_aux mlist =
  match mlist with
  |[]-> Noeud(true ,[])
  |t::q->Noeud(false,[t,creer_aux q])
  ;;
let creer m =
  let mlist = mot_of_string m in
  creer_aux mlist
  ;;
```

[4]:
```
val creer_aux : char list -> arbre_lex = <fun>
```

[4]:
```
val creer : string -> arbre_lex = <fun>
```

[5]:
```
creer "test";;
```

[5]:
```
- : arbre_lex =
Noeud (false,
 [('t',
   Noeud (false,
    [('e', Noeud (false, [('s', Noeud (false, [('t', Noeud (true,
[]))]))]))]))])
```

6 - On a :

[6]:
```
let rec compterAux a =
  match a with
  |Noeud(false,fils)->compter_fils fils
  |Noeud(true,fils)->1+compter_fils fils
and compter_fils l=
  match l with
  |[]->0
  |(c,a)::q->compterAux a + compter_fils q
;;
let compter a=
  match a with
  |Noeud(true,[]) ->0
  |_ -> compterAux a
;;
```

[6]:
```
val compterAux : arbre_lex -> int = <fun>
val compter_fils : fils -> int = <fun>
```

[6]:
```
val compter : arbre_lex -> int = <fun>
```

[7]:
```
let feuille= Noeud(true ,[]) ;;
let exemple=
Noeud(false ,[
'f',Noeud(false ,[
'a', Noeud(true ,[
('c', feuille)
]
);
 ('i', feuille)
 ])
 ]);;
compter exemple;;
```

[7]:
```
val feuille : arbre_lex = Noeud (true, [])
```

[7]:
```
val exemple : arbre_lex =
  Noeud (false,
   [('f',
     Noeud (false,
      [('a', Noeud (true, [('c', Noeud (true, []))]));
       ('i', Noeud (true, []))]))])
```

[7]:
```
- : int = 3
```

7 - On a :

[8]:
```
let rec compterFeuille a =
  match a with
  |Noeud(true,[])->1
  |Noeud(true,fils) | Noeud(false,fils) ->compterFeuille_fils fils
and compterFeuille_fils l=
  match l with
  |[]->0
  |(c,a)::q->compterFeuille a + compterFeuille_fils q
;;
```

[8]:
```
val compterFeuille : arbre_lex -> int = <fun>
val compterFeuille_fils : fils -> int = <fun>
```

[9]:
```
compterFeuille feuille;;
```

[9]:
```
- : int = 1
```

9 - On a :

[10]:
```
let rec prefixe x l =
  match l with
  |[]-> []
  |t::q -> (x::t)::(prefixe x q)
;;
```

[10]:
```
val prefixe : 'a -> 'a list list -> 'a list list = <fun>
```

[11]:
```
prefixe 'a' [['v';'i';'o';'n'];['r';'r';'i';'v';'e']] ;;
```

[11]:
```
- : char list list =
[['a'; 'v'; 'i'; 'o'; 'n']; ['a'; 'r'; 'r'; 'i'; 'v'; 'e']]
```

10 - On a :

[12]:
```
let rec extraire a =
  match a with
  |Noeud(true,[])->[[]]
  |Noeud(false,fils)->extraire_fils fils
  |Noeud(true,fils)->extraire_fils fils
and extraire_fils l=
  match l with
  |[]-> []
  |(c,a)::q->(prefixe c (extraire a))@extraire_fils q
;;
```

[12]:
```
val extraire : arbre_lex -> char list list = <fun>
val extraire_fils : fils -> char list list = <fun>
```

[13]:
```
extraire exemple;;
```

[13]:
```
- : char list list = [['f'; 'a'; 'c']; ['f'; 'i']]
```

11 - On a :

[14]:
```
let rec ajoute m a =
  match a with
  |Noeud(b,fils) -> Noeud(b,ajoute_fils m fils)
and ajoute_fils m l =
  match m,l with
  |[],_ -> l
  |t::q1, [] -> [(t,creer_aux q1)]
  |t::q1,(c,a1)::q2 when t=c -> (c,ajoute q1 a1)::q2
  |t::q1,(c,a1)::q2 when t<c -> (t,creer_aux q1)::l
  |t::q1,(c,a1)::q2 -> (c,a1)::(ajoute_fils m q2)
;;
```

[14]:
```
val ajoute : char list -> arbre_lex -> arbre_lex = <fun>
val ajoute_fils : char list -> fils -> fils = <fun>
```

12 - On a :

[15]:
```
let rec arbre_of_listAux listmot a =
  match listmot with
  |[]->a
  |t::q-> arbre_of_listAux q (ajoute (mot_of_string t) a)
  ;;
let arbre_of_list listmot=
  arbre_of_listAux listmot (Noeud(true ,[]))
  ;;
```

```
[15]: val arbre_of_listAux : string list -> arbre_lex -> arbre_lex = <fun>
```

```
[15]: val arbre_of_list : string list -> arbre_lex = <fun>
```

```
[16]: arbre_of_list ["avion";"attendre";"arrive";"test"];;
```

```
[16]: - : arbre_lex =
Noeud (true,
 [('a',
   Noeud (false,
    [('r',
      Noeud (false,
       [('r',
         Noeud (false,
          [('i',
            Noeud (false, [('v', Noeud (false, [('e', Noeud (true,
[]))]))]))]))]));
     ('t',
      Noeud (false,
       [('t',
         Noeud (false,
          [('e',
            Noeud (false,
             [('n',
               Noeud (false,
                [('d',
                  Noeud (false,
                   [('r', Noeud (false, [('e', Noeud (true,
[]))]))]))]))]))]))]));
     ('v',
      Noeud (false,
       [('i',
         Noeud (false, [('o', Noeud (false, [('n', Noeud (true,
[]))]))]))]))]));
  ('t',
   Noeud (false,
    [('e', Noeud (false, [('s', Noeud (false, [('t', Noeud (true,
[]))]))]))]))])
```

13 - On a :

[17]:
```
let lecture nom_fichier =
  let f = open_in nom_fichier in
  let rec lecture_rec l =
    try
      lecture_rec ((input_line f)::l);
    with
      |End_of_file -> arbre_of_list l
    in lecture_rec [] ;;
```

[17]:
```
val lecture : string -> arbre_lex = <fun>
```

Le fichier texte contient trop de mot et on obtient l'erreur : *"Stack overflow during evaluation (looping recursion?)."* Il faudrait peut être agrandir la pile (?) ou réduire le nombre de mot car la fonction est fonctionnelle avec un fichier avec moins de mots