TD3 : Arbres lexicographiques

▷ **Support étudiant**

Un arbre lexicographique (ou "trie") est une structure utilisée pour représenter des ensembles dont les éléments admettent une décomposition séquentielle en "caractères" possédant de plus un ordre (par exemple, les entiers admettent une décomposition en suite de chiffres, les chaînes de caractères en suite de caractères, etc).

Un arbre lexicographique contient :

- une fonction de décomposition permettant de transformer un élément en liste de "caractères" ;
- une fonction de recomposition, inverse de la précédente ;
- un arbre n-aire dont les branches représentent les suites constituant les éléments stockés dans l'ensemble, les "caractères" étant rangés dans les branches.

De plus, les branches sont triées de gauche à droite. Pour chaque nœud, il doit être indiqué si la suite lue depuis la racine constitue la décomposition d'un élément ou seulement un préfixe strict d'une telle décomposition.

Pour simplifier l'écriture des algorithmes, il vous est demandé de ranger **les "caractères" dans les branches**. Un exemple d'arbre lexicographique stockant le long des branches les mots du mini-dictionnaire suivant : *bas, bât, de, la, lai, laid, lait, lard, le, les, long* est présenté Fig. 1.

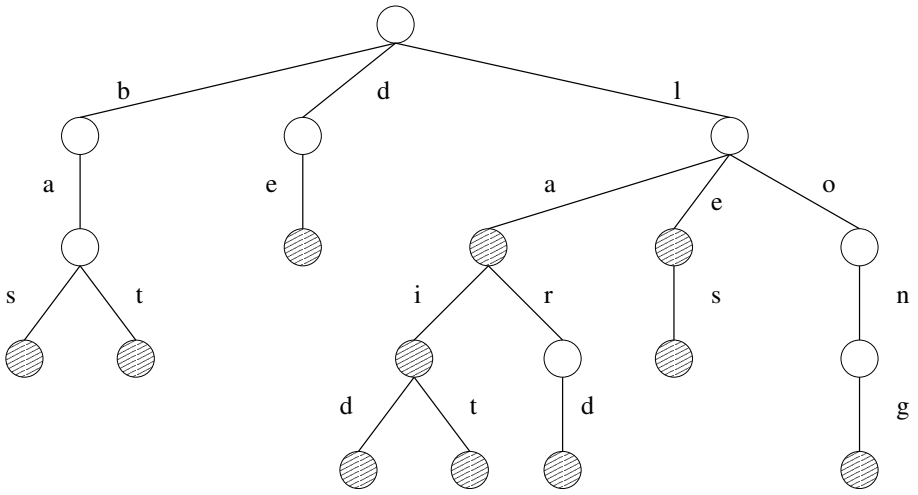

Figure 1 – Un mini-dictionnaire.

1 Structure arborescente n-aire

Dans un premier temps, nous nous intéresserons juste à la structure arborescente n-aire.

▷ **Exercice 1 (Définition des types)** *Définir le type* 'a arbre *représentant la structure arborescente : arbre n-aire avec les booléens dans les nœuds et des* 'a *dans les branches. Vous pouvez également définir un second type représentant les branches.*

▷ ***Solution***

```
type 'a arbre = Noeud of bool * ('a branche list)
and 'a branche = 'a * 'a arbre
```

▷ **Exercice 2 (Test d'appartenance)** *Écrire la fonction* appartient *qui teste si un élément appartient bien à un ensemble représenté par un arbre lexicographique.*

▷ ***Solution***

```
val appartient : 'a list -> 'a arbre -> bool = <fun>
```

Ils risquent d'écrire quelque chose comme ça :

```
let rec appartient_naif lc (Noeud (b,lb)) =
  match lc,lb with
      (* on a épuisé la liste : le résultat est le booléen du noeud sur
        lequel on est arrivé *)
      [], _ -> b
      (* sinon on cherche la branche correspondant au premier
        caractère de la liste :
          - elle n'existe pas : le mot n'appartient pas au trie
          - on la trouve, on relance aux avec le reste de la liste
            et l'arbre de cette branche *)
    | _,[] -> false (* l'arbre est vide, le mot n'appartient pas *)
    | c::qlc,(c1,b1)::qlb ->
        if c=c1
        then appartient_naif qlc b1 (* on a la branche *)
        else if c < c1
             then false (* la branche n'existera pas *)
             else appartient_naif lc (Noeud (b,qlb)) (* on continue à chercher notre branche *)
```

Il faut les amener à trouver qu'ils ont besoin d'une fonction de recherche de branche sur les arbres. Leur faire écrire les commentaires sur à quoi correspond chaque cas, peut leur faire remarquer qu'ils cherchent s'il existe une branche qui commence par le caractère 'c'.

```
(*****************************************************************************)
(*   fonction interne au Module de recherche dans une liste                  *)
(*   de la branche correspondant à un caractère                              *)
(*   signature : recherche :                                                 *)
(*                 'a -> ('a * 'b) list -> ('a * 'b) option = <fun>          *)
(*   paramètres : - un caractère                                             *)
(*                - une liste de branches                                    *)
(*   résultat   : Some (la branche correspondant au caractère),              *)
(*                si elle existe                                             *)
(*                None, sinon                                                *)
(*****************************************************************************)
let rec recherche c lb =
  match lb with
    | []                          -> None
    | (tc, ta)::qlb -> if (c < tc)
                       then None
                       else if (c = tc)
                            then Some ta
                            else recherche c qlb
(* TEST *)
let%test _ = (recherche 'b' b1 = Some bb)
let%test _ = (recherche 'd' b1 = Some bd)
let%test _ = (recherche 'l' b1 = Some bl)
let%test _ = (recherche 'a' b1 = None)
```

```
let rec appartient lc (Noeud (b,lb)) =
  match lc with
      (* on a épuisé la liste : le résultat est le booléen du noeud sur
         lequel on est arrivé *)
      [] -> b
      (* sinon on cherche la branche correspondant au premier
         caractère de la liste :
           - elle n'existe pas : le mot n'appartient pas au trie
           - on la trouve, on relance aux avec le reste de la liste
             et l'arbre de cette branche *)
    | c::qlc -> match (recherche c lb) with
                  | None   -> false
                  | Some a -> appartient qlc a
```

Bien leur faire remarquer que le type de recherche *est* **val** recherche : 'a -> ('a * 'b) list -> 'b option = <**fun**> *et donc travaille sur une table associative, complètement indépendamment de notre problème d'arbre.*

▷ **Exercice 3 (Ajout)** *Écrire la fonction* ajout *qui ajoute un élément à un ensemble représenté par un arbre lexicographique.*

▷ ***Solution***

```
val ajout : 'a list -> 'a arbre -> 'a arbre = <fun>
```

S'ils n'ont pas retenu la leçon de la fonction précédente, ils risquent d'écrire quelque chose comme ça :

```
let rec ajout_naif lc (Noeud (b,lb)) =
match lc,lb with
      (* on a épuisé la liste : le résultat est le noeud sur lequel on
         est arrivé avec son booléen mis à vrai *)
    [], _ -> Noeud (true,lb)
  | c::qlc,[] ->
      (* l'arbre est vide, il faut créer récursivement la branche *)
      Noeud (b,[(c,ajout_naif qlc (Noeud(false,[])))])
  | c::qlc,(c1,b1)::qlb ->
      if (c=c1)
      then Noeud (b,(c,ajout_naif qlc b1)::qlb) (* on a trouvé la branche, on continue dans celle-ci *)
      else if (c>c1)
           then (* on conitnue à chercher la branche *)
           let (Noeud (baj,lbaj)) = ajout_naif lc (Noeud (b,qlb)) in
           Noeud (b,(c1,b1)::lbaj)
           else (* on ne trouvera jamais la branche, on la créer *)
           Noeud (b,(c,ajout_naif qlc (Noeud(false,[])))::lb)
```

Il faut les amener à se rendre compte qu'ils peuvent ré-utiliser la fonction de recherche précédente, et qu'ils ont besoin d'une fonction de mise à jour d'une branche.

```
(*************************************************************************)
(*   fonction interne au Module d'ajout/mise àjour du fils d'un arbre    *)
(*   signature : maj :                                                   *)
(*                'a -> 'a * 'b -> ('a * 'b) list -> ('a * 'b) list = <fun> *)
(*   paramètres : - un caractère                                         *)
(*                - la branche à ajouter/modifier                        *)
(*                - la liste de branches                                 *)
(*   résultat : la liste de branches mise à jour                         *)
(*************************************************************************)
```

```
let rec maj c nouvelle_b lb =
  match lb with
   | []                              -> [c, nouvelle_b]
   | (tc, ta)::qlb -> if (c < tc)
                      then (c, nouvelle_b)::lb
                      else if (c = tc)
                           then (c, nouvelle_b)::qlb
                            else (tc, ta)::( maj c nouvelle_b qlb)

(* TESTS *)
let%test _ = (maj 'b' bl b1 = [bl;bd;bl])
let ba = ('a',Noeud(true,[('n',Noeud(true ,[]));(' u',Noeud(true ,[]))]))
let%test _ = (maj 'a' ba b1 = [ba;bb;bd;bl])
let bm = ('m',Noeud(false,[('a',Noeud(true ,[]))]))
let%test _ = (maj 'm' bm b1 = [bb;bd;bl;bm])

let rec ajout lc (Noeud (b, lb)) =
  match lc with
      (* on a épuisé la liste : le résultat est le noeud sur lequel on
         est arrivé avec son booléen mis à vrai *)
    | [] -> Noeud (true, lb)
      (* sinon on cherche l'arbre arbre_c de la branche correspondant
         au premier caractère de la liste ;
         si on ne le trouve pas, le résultat de cette recherche est un arbre
         avec une liste de branches vide.

         Le résultat de aux est le noeud en paramètre
         que l'on met à jour en remplacant dans sa liste de branches,
         la branche du premier caractère par la branche dont l'arbre est
         le résultat de l'ajout du reste des caractères à l'arbre arbre_c
       *)
    | c::qlc -> let arbre_c =
                  match (recherche c lb) with
                    | None   -> Noeud (false, [])
                    | Some a -> a
                in Noeud (b, maj c (ajout qlc arbre_c) lb)
```

Bien leur faire remarquer que le type de maj *est* **val** maj : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <**fun**> *et donc travaille sur une table associative, complètement indépendamment de notre problème d'arbre.*

On peut donc décomposer notre problème sur 3 niveaux :

- *ce qui concerne les branches vu comme une table associative (*recherche *et* maj*)*
- *ce qui concerne la structure arborescente (*appartient *et* ajout*)*
- *ce qui concerne les arbres lexicographiques (ce qui suit :* appartient_trie *et* ajout_trie*)*

▷ **Support étudiant**

2 Arbres lexicographiques

En plus de la structure arborescente, un arbre lexicographique contient une fonction de décomposition et une fonction de recomposition.

Par exemple, dans le cas d'un dictionnaire de mot, la fonction de décomposition sera de type string -> char list et la fonction de recomposition de char list -> string, permettant par exemple de passer de "laid" à ['l';'a';'i';'d'].

▷ **Exercice 4** *Définir le type* ('a,'b) trie *permettant de représenter des arbres lexicographiques (arbre+fonction de décomposition+fonction de recomposition).*

▷ ***Solution***

```
type ('a,'b) trie = Trie of 'b arbre * ('a -> 'b list) * ('b list -> 'a)
```

▷ **Exercice 5 (Test d'appartenance)** *Écrire la fonction* appartient_trie *qui teste si un élément appartient bien à un ensemble représenté par un arbre lexicographique.*

▷ ***Solution***

```
val appartient_trie : 'a -> ('a, 'b) trie -> bool = <fun>

let appartient_trie mot (Trie (a,decomp,comp)) =
  let lc = decomp mot in
  appartient lc a
```

▷ **Exercice 6 (Ajout)** *Écrire la fonction* ajout_trie *qui ajoute un élément à un ensemble représenté par un arbre lexicographique.*

▷ ***Solution***

```
val ajout_trie : 'a -> ('a, 'b) trie -> ('a, 'b) trie = <fun>

let ajout_trie mot (Trie (a, decomp, comp)) =
  let lc = decomp mot in
  Trie (ajout lc a, decomp, comp)
```