NFP119: feuille d'exercices 5

María-Virginia Aponte

2008

Exercice 1

Considérez le type `'a arbre` étudié en cours. Écrivez les fonctions suivantes.

1. `profondeur` La profondeur d'un arbre est 0 s'il est vide, et égale à la taille (nombre de noeuds) de la branche la plus longue sinon.

Solution:

```
type 'a arbre = Vide
              | Noeud of 'a arbre * 'a * 'a arbre;;
# let rec profondeur = function
    Vide →0
  | Noeud (g,_,d) →1+ max (profondeur g) (profondeur d);;
val profondeur : 'a arbre →int = <fun>

# profondeur (Noeud (Noeud (Vide, 1, Vide), 3, Vide));;
- : int = 2
```

2. `existe_arbre` Teste si au moins un des éléments de l'arbre satisfait une condition passée en argument.

Solution:

```
# let rec existe cond = function
    Vide →false
  | Noeud (g,x,d) →cond x || existe cond g || existe cond d;;
    val existe : ('a →bool) →'a arbre →bool = <fun>

# existe (fun x →x mod 2 = 0) (Noeud (Noeud (Vide, 1, Vide), 3, Vide));;
- : bool = false
```

3. `map_arbre f a` Renvoie un nouvel arbre obtenu par application de la fonction `f` sur chaque noeud de l'arbre a.

Solution:

```
# let rec map_arbre f = function
    Vide →Vide
  | Noeud (g,x,d) →let g' = map_arbre f g and d' = map_arbre f d in
                   Noeud(g', (f x), d');;
val map_arbre : ('a →'b) →'a arbre →'b arbre = <fun>

# map_arbre (fun x →string_of_int x) (Noeud (Noeud (Vide, 1, Vide), 3, Vide));;
- : string arbre = Noeud (Noeud (Vide, "1", Vide), "3", Vide)
```

4. `sym_arbre` Renvoie l'arbre symétrique d'un arbre.

Solution:

```
let rec sym_arbre a =
  match a with
    | Vide →Vide
    | Branche(i,fg,fd) →Branche(f i , sym_arbre fd, sym_arbre fg)
```

Exercice 4

On se donne un type de donnée (voir page suivante) permettant de représenter un dictionnaire (non accentué) sous la forme d'un arbre dont tous les chemins de la racine aux feuilles représentent un mot du dictionnaire. Les nœuds de cet arbre ont un nombre variable de sous-arbres. Si un nœud a zéro sous-arbres, il s'agit d'une feuille. On se donne aussi un dictionnaire de test contenant les chemins suivants : `ARBRE AUX AS BELLE BEC`. Voir la page suivante.

1. Écrire la fonction `dico_assoc` définie comme suit : `dico_assoc c d` rend le `noeud` du dictionnaire `d` ayant le caractère `c` au sommet. On rappelle que la fonction `List.find : ('a -> bool) -> 'a list -> 'a` permet de trouver un élément dans une liste (et lève l'exception `Not_found` en cas d'échec). `dico_assoc` lèvera l'exception `Not_found` en cas d'échec.
2. Écrire la fonction `existe` définie comme suit : `existe d m` rend true si la chaîne de caractère m appartient au dictionnaire d. Il existe plusieurs méthodes : transformation préalable de la chaîne en liste de caractères, utilisation d'un indice pour tester le bon caractère dans la chaîne... On pourra utiliser la fonction `String.get` permettant d'accéder aux $i^{\text{ème}}$ caractère d'une chaîne.
3. Modifier la structure de donnée afin de "marquer" les fins de mots possibles (on n'acceptera pas `ARB`, mais on acceptera `BEL`). Reprogrammez `existe`.
4. Écrire une fonction `ajoute` qui prend une chaîne de caractères et l'ajoute à un dictionnaire.

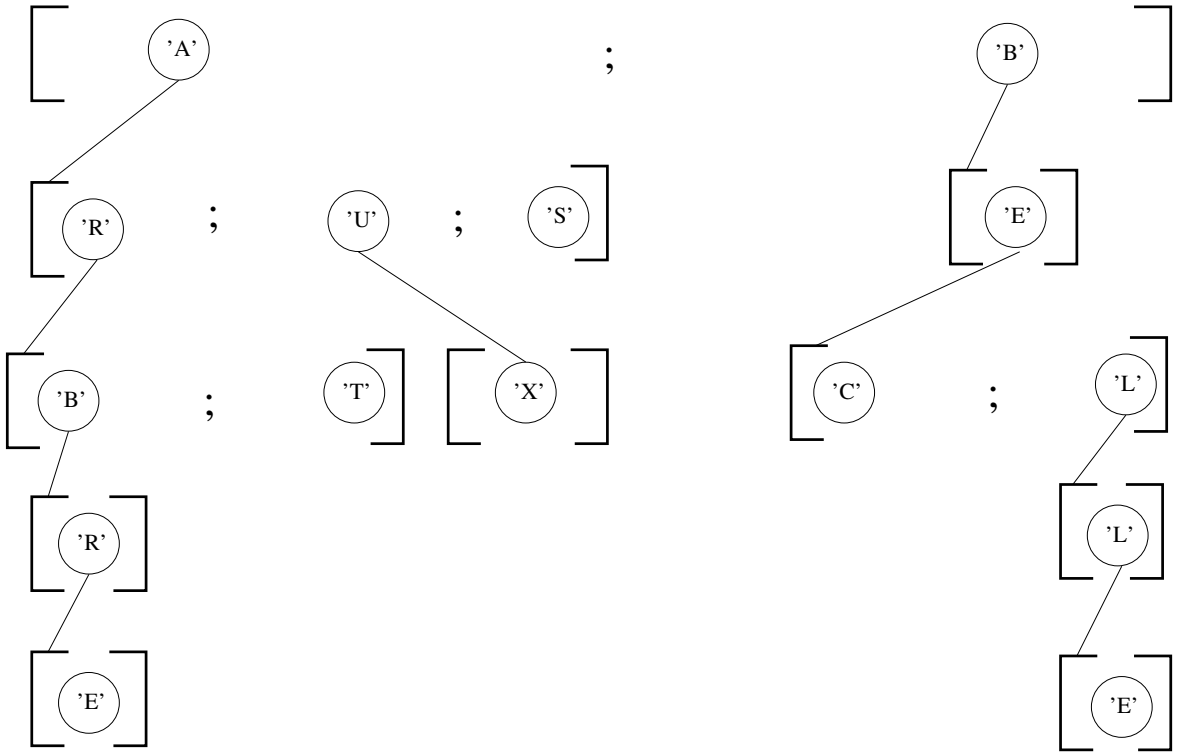
```
type noeud = Node of char * noeud list
type dico = noeud list

let d7:dico = [Node('t',[])]
let d6:dico = [Node('c',[]);Node('l',[Node('l',[Node('e',[])])])]
let d5:dico = [Node('e',d6)]
let d4:dico = [Node('x',[])]
let d3:dico = [Node('b',[Node('r',[Node('e',[])])])]
let d2:dico = [Node('r',d3@d7);Node('u',d4);Node('s',[])]
let d1:dico = [Node('a',d2);Node('b',d5)]
```

Ce qui donne la valeur suivante pour `d1` :

```
d1=[Node ('a',
     [Node ('r', [Node ('b', [Node ('r', [Node ('e', [])])]);
                  Node ('t', [])]);
      Node ('u', [Node ('x', [])]);
      Node ('s', [])]);
    Node ('b',
     [Node ('e', [Node ('c', []); Node ('l', [Node ('l', [Node ('e', [])])])])])]
```

Et `d1` correspond au schéma suuivant :



Solution:

On rappel que comme le type noeud n'a qu'un seul constructeur, on peut écrire : **match** `x` **with** `Node(c,nl)` →`e` de la manière suivante : **let** `Node(c,nl) = x` **in** `e`. *)

Remarque : dico_assoc n'a pas besoin d'être définie en même temps que existe_i, car elle n'a pas besoin d'appeler existe_i. Si elle en avait besoin, on les aurait défini en même temps à l'aide de and comme suit :

```
let rec dico_assoc ... =
  ... existe_i ...
and existe_i ... =
  ... dico_assoc ...
```

```
type noeud = Node of char * noeud list
type dico = noeud list

(** [dico_assoc c d] rend le noeud du dictionnaire [d] ayant le
    caractère [c] au sommet.
    @raise Not_found si un tel noeud n'existe pas dans [d]. *)
let dico_assoc (c:char) (d:dico) : noeud =
  List.find (fun nd →let Node (c',_) = nd in c=c') d

(** [existe_i d m i] rend true si la chaîne de caractère [m] *à partir
    du [i]eme caractère* appartient au dictionnaire [d]. Il s'agit
    d'une fonction auxiliaire à existe. On aurait pu la définir
    localement à existe. *)
let rec existe_i (d:dico) (m:string) (i:int) : bool =
  try
    let c = String.get m i in
    let nd_avec_c_au_sommet = dico_assoc c d in
    let Node(_,sous_dico) = nd_avec_c_au_sommet in
    existe_i sous_dico m (i+1)
  with
    | Not_found →false (* on n'a pas trouvé de noeud avec c au sommet. *)
    | Invalid_argument _ →true (* On est arrivé au bout du mot. *)

(** [existe d m] rend true si la chaîne de caractère [m] appartient au
    dictionnaire [d]. *)
let existe (d:dico) (m:string) : bool =
  existe_i d m 0 (* On se content d'appeler existe_i en commençant à 0. *)

(* TESTS *)
let d7:dico = [Node('t',[])]
let d6:dico = [Node('c',[]);Node('l',[Node('l',[Node('e',[])])])]
let d5:dico = [Node('e',d6)]
let d4:dico = [Node('x',[])]
let d3:dico = [Node('b',[Node('r',[Node('e',[])])])]
let d2:dico = [Node('r',d3@d7);Node('u',d4);Node('s',[])]
let d1:dico = [Node('a',d2);Node('b',d5)]
existe d1 "aux";;
existe d1 "auxi";;
existe d1 "au";;
existe d1 "bell";;
existe d1 "belle";;
existe d1 "bclle";;
```