Épreuve finale

Janvier 2021

Documents autorisés: poly et notes de cours, notes de TD

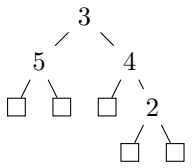
L'épreuve durera 1 heure et 30 minutes. Le sujet vaut 35 points plus 3 points de bonus.

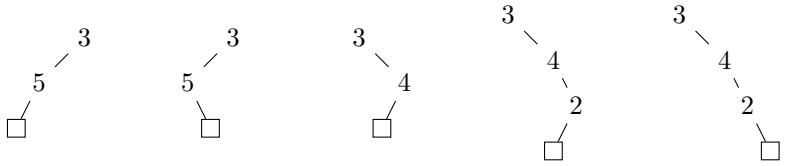
On rappelle le type utilisé dans le cours pour représenter les arbres binaires

```
type 'a btree =
    Empty
  | Node of 'a * 'a btree * 'a btree
```

Exercice I : Arbres binaires 1

Branches On appelle *branche* d'un arbre binaire la liste des étiquettes que l'on rencontre en «descendant» dans l'arbre depuis la racine jusqu'à un arbre vide. À chaque étape de la descente, on choisit de «descendre» à gauche ou à droite.

Par exemple dans l'arbre 

on a les branches 

qui correspondent aux listes `[3;5]`; `[3;4]` et `[3;4;2]`. Notez que certaines listes peuvent correspondre à deux branches.

Q1 – (1pt) Définir la fonction de signature

```
cons_all (x:'a) (xss:('a list) list) : ('a list) list
```

qui donne la liste de listes obtenue en ajoutant `x` en tête de chaque liste de `xss`.

Schématiquement: `(cons_all x [xs1; ..; xsn])`=`[x::xs1; ..; x::xsn]`

Bonus: +1pt pour l'utilisation d'un itérateur.

Q2 – (2pts) En utilisant `cons_all` définir la fonction

```
branch_list (bt:'a btree) : ('a list) list
```

qui donne la liste de toutes les branches de `bt`.

Rappel: l'arbre vide `Empty` contient une branche correspondant à la liste vide `[]`.

Sur l'arbre dessiné ci-dessus, on aura la liste de listes: `[[3; 5]; [3; 5]; [3; 4]; [3; 4; 2]; [3; 4; 2]]`

Q3 – (3pts) Définir la fonction de signature

```
is_branch (xs:'a list) (bt:'a btree) : bool
```

qui donne la valeur `true` si la liste `xs` correspond à une branche de `bt`; et `false` sinon.

Exemples: appelons `bt` l'arbre dessiné ci-dessus, on a
`(is_branch [3] bt)` = `false`
`(is_branch [3;4] bt)` = `true`
`(is_branch [3;2] bt)` = `false`
`(is_branch [3;4;2] bt)` = `true`
`(is_branch [3;2;4] bt)` = `false`
`(is_branch [3;4;2;1] bt)` = `false`

On a également que `(is_branch [] Empty)` = `true`.

Solution Exercice I

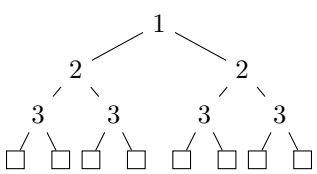
```
type 'a btree =
    Empty
  | Node of 'a * 'a btree * 'a btree

(* Q1 *)
let rec cons_all (x:'a) (xss:('a list) list) : ('a list) list =
  match xss with
      [] -> []
    | xs::xss -> (x::xs)::(cons_all x xss)

let cons_all (x:'a) (xss:('a list) list) : ('a list) list =
  List.map (fun xs -> x::xs) xss

(* Q2 *)
let rec branch_list (bt:'a btree) : ('a list) list =
  match bt with
      Empty -> [[]]
    | Node(x,bt1,bt2) ->
          (cons_all x ((branch_list bt1)@(branch_list bt2)))

(* Q3 *)
let rec is_branch (xs:'a list) (bt:' btree) : bool =
  match xs, bt with
      [], Empty -> true
    | x::xs, Node(y,bt1,bt2) ->
        (x=y) && ((is_branch xs bt1) || (is_branch xs bt2))
    | _ -> false
```

Exercice II : Arbres binaires 2

Sous-listes On appelle ici *sous-listes* d'une liste `xs` toutes les listes contenant des éléments de `xs` dans l'ordre où ils apparaissent dans `xs`.

Par exemple, la liste `[1;2;3]` a pour sous-listes les listes `[1]`, `[1;2]`, `[1;3]`, `[1;2;3]`, `[2]`, `[2;3]`, `[3]` et `[]`.

On peut utiliser les structures d'arbres binaires pour calculer l'ensemble des sous-listes d'une liste. Par exemple, pour la liste `[1;2;3]` on construit l'arbre :

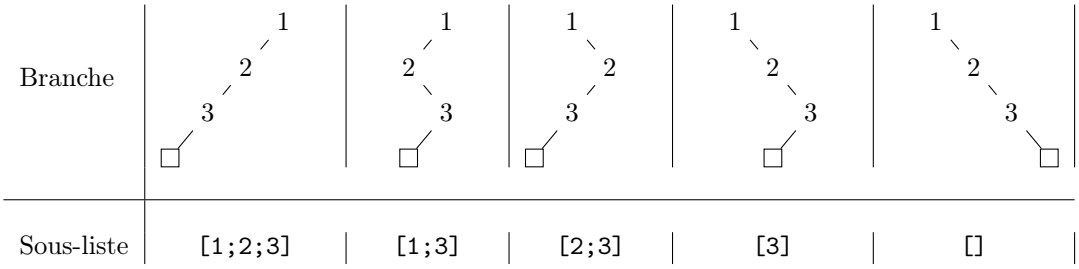


On dira qu'un tel arbre est un *arbre complet* pour la liste `[1;2;3]`.

Dans un arbre complet pour une liste `xs`, chaque branche correspond à la liste `xs`. Pour obtenir les sous-listes à partir de

ces branches, on «interprète» un branchement à gauche comme la présence de la valeur de l'étiquette dans la sous-liste et, un branchement à droite, comme son absence de la sous-liste.

Exemples:



Q1 – (2pts) Définir la fonction

`sublist_tree (xs:'a list) : 'a btree`

qui construit l'arbre complet pour `xs`.

Q2 – (2pts) Définir la fonction

`sublist_list (bt:'a btree) : ('a list) list`

telle que si `bt` est un arbre complet (pour une liste `ys`) alors `(sublist_list bt)` donne la liste des sous-listes de `ys`.

Remarque: utilisez la fonction `cons_all` de l'exercice précédent.

Q3 – (3pts) Définir la fonction

`is_sublist (xs:'a list) (bt:'a btree) : bool`

telle que si `bt` est un arbre complet (pour une liste `ys`) alors `(is_sublist xs bt)`=`true` si et seulement si `xs` est une sous-liste de `ys`. En particulier, `(is_sublist [] Empty)`=`true`.

Solution Exercice II

```
type 'a btree =
    Empty
  | Node of 'a * 'a btree * 'a btree

(* Q1 *)
let rec sublist_tree (xs:'a list) : 'a btree =
  match xs with
      [] -> Empty
    | x::xs -> let bt = sublist_tree xs in
               Node (x, bt, bt)

(* Q2 *)
let rec sublist_list (bt:'a btree) : ('a list) list =
  match bt with
      Empty -> [[]]
    | Node(x,bt1,bt2) ->
        (cons_all x (sublist_list bt1))@(sublist_list bt2)

(* Q3 *)
let rec is_sublist (xs:'a list) (bt:' btree) : bool =
  match xs, bt with
      [], _ -> true
    | x::xs', Node(y,bt1,bt2) ->
        if (x=y) then (is_sublist xs' bt1)
        else (is_sublist xs bt2)
    | _ -> false
```

Exercice III : Listes 1

On dispose d'une liste de triplets donnant la distance (à vol d'oiseau) entre deux villes.
Les triplets sont de type `(string * string * float)`. Par exemple `("Paris","Marseille",660.91)`.

Posons: `type dist_list = (string * string * float) list`

On supposera que les listes de distances ne contiennent pas d'informations incohérentes. Par exemple, ne contiennent pas un premier triplet `("Paris", "Marseille", 660.91)` et un autre `("Marseille", "Paris", 657.78)`.

On peut utiliser les listes de distances pour obtenir la distance entre deux villes.

Q1 – (3pts) Définir la fonction

`dist (v1:string) (v2:string) (ds:dist_list) : float`

qui donne la distance entre `v1` et `v2` indiquée dans la liste de distances `ds`. La fonction déclenche l'exception `Not_found` si aucun triplet de `ds` ne donne la distance entre `v1` et `v2`.

Prenez garde qu'il faut avoir que `(dist "Paris" "Marseille" [("Paris","Marseille",660.91)]` donne 660.91, mais aussi que `(dist "Marseille" "Paris" [("Paris","Marseille",660.91)]` donne aussi 660.91.

Double liste d'association On peut améliorer un peu la recherche de la distance entre deux villes en utilisant un structure un peu plus complexe: une *table des distances* représentée comme une liste d'association dont les clefs sont des noms de ville et les valeurs associées des listes d'association entre ville et distance. Par exemple, la liste de distances `[("Paris","Marseille",660.91)]` est réprésentée par la table de distances `[("Paris",[("Marseille",660.91)]); ("Marseille",[("Paris",660.91)])]`. Notez comment l'information est dupliquée.

Autre exemple, la liste de distances `[("Paris","Marseille",660.91); ("Marseille","Toulouse",319.79)]` est représentée par la table de distances

```
[ ("Paris",[("Marseille",660.91)]);
  ("Marseille",[("Paris",660.91); ("Toulouse",319.79)]);
  ("Toulouse",[("Marseille",319.79)]) ]
```

On pose: `type dist_tab = (string * (string * float) list) list`

Q2 – (1pt) En utilisant la fonction `List.assoc` de la biblothèque standard, et une table de distances, redéfinir la fonction

`dist (v1:string) (v2:string) (tds : dist_tab) : float`

qui donne la distance entre les villes `v1` et `v2` telle qu'indiquée dans la table de distances `tds`. Ne vous occupez pas du cas où l'information ne figure pas dans la table `tds`, `List.assoc` s'en chargera.

Nous allons maintenant transformer une *liste de distances* en *table de distances*.

Q3 – (3pts) Pour cela, on définit au préalable une fonction générique d'ajout dans une liste d'association de type (générique) `('a * 'b list) list`. C'est-à-dire que l'on associe à une clef une listes de valeurs.

Définir la fonction

`add (k:'a) (v:'b) (kvss:('a * 'b list) list) : ('a * b' list) list`

qui ajoute à la liste d'association `kvss` la valeur `v` à la liste des valeurs associées à la clef `k`. Si aucune association n'existe pour la clef `k` dans `kvss` alors la fonction ajoute la liaison `(k,[v])` à `kvss`.

Exemples:
`(add 'x' 42 [])` = `[('x',[42])]`
`(add 'x' 42 [('x',[43])]); ('y',[1;2])` = `[('x',[42;43]); ('y',[1;2])]`
`(add 'x' 42 [('y',[1;2]); ('x',[43])])` = `[('y',[1;2]); ('x',[42;43])]`

On ne cherchera pas à gérer les duplications de valeurs dans les listes associées. Par exemple:
`(add 'x' 42 [('x',[42])])` = `[('x',[42;42])]`

Q4 – (1pt) Utilisez la fonction `add` pour définir la fonction

`add_dist (v1:string) (v2:string) (d:float) (tds:dist_tab) : dist_tab`

qui donne la table de distances à laquelle on a ajouté l'information de distance `d` entre les villes `v1` et `v2`.

Exemples:

```
(add_dist "Paris" "Marseille" 660.91 [])
```
donne la liste `[("Paris",[("Marseille",660.91)]); ("Marseille",[("Paris",660.91)])]`

```
(add_dist "Toulouse" "Marseille" 319.79
          [("Paris",[("Marseille",660.91)]);
           ("Marseille",[("Paris",660.91)])])}
```

donne la liste

```
  [ ("Paris",[("Marseille",660.91)]);
    ("Marseille",[("Paris",660.91); ("Toulouse",319.79)]);
    ("Toulouse",[("Marseille",319.79)]) ]
```

L'ordre des éléments dans les listes est indifférent.

Q5 – (2pt) Définir la fonction

`build (ds: dist_list) : dist_tab`

qui transforme la liste de distances `ds` en une table de distances.

Bonus: +1pt pour une version récursive terminale.

Solution Exercice III

```
type dist_list = (string * string * float) list

(* Q1 *)
let rec dist (v1:string) (v2:string) (ds:dist_list) : float =
  match ds with
      [] -> raise Not_found
    | (w1,w2,d)::ds ->
        if ((w1=v1) && (w2=v2)) || ((w1=v2) && (w2=v1)) then d
        else (dist v1 v2 ds)

type dist_tab = (string * (string * float) list) list

(* Q2 *)
let dist (v1:string) (v2:string) (tds:dist_tab) : float =
  List.assoc v2 (List.assoc v1 tds)

(* Q3 *)
let rec add (k:'a) (v:'b) (kvss:('a * 'b list) list) : ('a *'b list) list =
  match kvss with
      (k',vs)::kvss' ->
        if (k=k') then (k, v::vs)::kvss'
        else (k',vs)::(add k v kvss')
    | [] -> [k,[v]]

(* Q4 *)
let add_dist (v1:string) (v2:string) (d:float) (tds:dist_tab) : dist_tab =
  (add v1 (v2,d) (add v2 (v1,d) tds))

(* Q5 *)
let rec build (ds: dist_list) : dist_tab =
```

```
  match ds with
      [] -> []
    | (v1,v2,d)::ds -> (add_dist v1 v2 d (build ds))

(* recursive terminale *)
let build (ds: dist_list) : dist_tab =
  let rec loop (ds:dist_list) (r:dist_tab) : dist_tab =
    match ds with
        [] -> r
      | (v1,v2,d)::ds -> (loop ds (add_dist v1 v2 d r))
  in (loop ds [])

(* iterateur *)
let build (ds: dist_list) : dist_tab =
  List.fold_left (fun r (v1,v2,d) -> (add_dist v1 v2 d r)) [] ds
```

Exercice IV : Listes 2

On reprend dans cet exercice (dernière question) le type `dist_tab` défini à l'exercice précédent.

Dans cet exercice, on utilise les fonctions `List.mem` et `List.assoc` de la bibliothèque standard.

Étant donné une ville de départ `v`, une liste de villes `vs` à visiter et une table de distances entre villes `vds`, on calcule un itinéraire (c'est-à-dire, une liste de villes) selon l'algorithme récursif suivant:

si `vs` est vide le résultat est la liste `[v]`
sinon
  choisir `v'` dans `vs` de distance minimale avec `v`
  placer `v` en tête du résultat
  recommencer avec `v'` et la liste `vs` dont on a supprimé `v'`

Q1 – (2pts) Définir la fonction

`list_remove (x:'a) (xs:'a list) : 'a list`

qui donne la liste obtenue en supprimant de la liste `xs` la première occurrence de `x` lorsqu'elle existe. Ainsi, si `x` est présent plusieurs fois dans `xs`, une seule occurence de `x` est supprimée.

Q2 – (2pts) Définir la fonction

`sub_assoc (xs:'a list) (xvs:('a * 'b) list) : ('a * 'b) list`

qui donne la sous-liste de `xvs` dont les clefs sont dans `xs`.

Par exemple: `(sub_assoc ["v1";"v3"] [("v1",234.78); ("v2",567.23); ("v3",345.12)])` donne `[("v1",234.78); ("v3",345.12)]`.

Bonus: +1pt pour l'utilisation d'un itérateur

Q3 – (4pts) Définir la fonction

`min_val_key (kvs:('a * 'b) list) : 'a`

qui donne la clef associée à la valeur minimale de `kvs`. Par exemple, `min_val_key` appliquée à `[("Paris",660.91); ("Toulouse",319.79)]` donne `"Toulouse"`. Si plusieurs valeurs sont minimales; on prend n'importe laquelle. Si `kvs` est vide, la fonction déclenche l'exception `(Invalid_argument "min_val_key")`.

Q4 – (4pts) Définir la fonction

`sort_dist (v:string) (vs:string list) (vds:dist_tab) : string list`

qui calcule la liste de villes donnée par l'algorithme défini au début de cet exercice.

Indication: pour *«choisir `v'` dans `vs` de distance minimale avec `v`»* pensez à `sub_assoc` et `min_val_key`.

Solution Exercice IV

```
(* Q1 *)
let rec list_remove (x:'a) (xs:'a list) : 'a list =
  match xs with
      [] -> []
    | x'::xs ->
        if (x=x') then xs
        else x'::(list_remove x xs)

(* Q2 *)
let rec sub_assoc (xs:'a list) (xvs:('a * 'b) list) : ('a * 'b) list =
  match xvs with
      [] -> []
    | (x,v)::xvs ->
        if (List.mem x xs) then (x,v)::(sub_assoc xs xvs)
        else (sub_assoc xs xvs)

(* itérateur *)
let sub_assoc (xs:'a list) (xvs:('a * 'b) list) : ('a * 'b) list =
  List.filter (fun (x,_) -> List.mem x xs) xvs

(* Q3 *)
let min_val_key (vds : ('a * 'b) list) : 'a =
  let rec loop (v':'a) (d:'b) (vds:('a * 'b) list) : 'a =
    match vds with
        [] -> v'
      | (v'',d')::vds ->
          if (d' < d) then (loop v'' d' vds)
          else (loop v' d vds)
  in match vds with
        [] -> raise (Invalid_argument("min_val_key"))
      | (v',d)::vds -> (loop v' d vds)

(* Q4 *)
let rec sort_dist (v:string) (vs:string list)
              (map : (string * (string * int) list) list)
    : string list =
  match vs with
      [] -> [v]
    | vs ->
          let v' = min_val_key (sub_assoc vs (List.assoc v map)) in
          v::(sort_dist v' (list_remove v' vs) map)
```