TD2 - Solution

Yaëlle Vinçont

1 Permutations

Exercice 1 Avec `List.map`, écrire `insert : 'a -> 'a list -> 'a list list` tq `insert x [v1; ...;vk]` renvoie une listes avec toutes les listes que l'on peut obtenir en insérant `x` dans `[v1; ...; vk]`.

Exemple `insert 1 [2;3] -> [ [1; 2; 3]; [2; 1; 3]; [2; 3; 1] ]`

Solution Idée de l'algorithme pour inserer x dans l :

— si la liste est vide, on a une seule possibilite d'insertion de x : `[[x]]`.

— sinon on a x à insérer dans hd : :tl : on insere x dans tl, puis on rajoute hd devant chaque possiblité. Plus le cas ou x est en tete de l. Par exemple :

```
/* cas ou x = 1, l = [3;4], hd = 3 et tl = [4] */
insere 1 [3;4]
/* on insere x dans tl */
r = insere 1 [4] = [ [1; 4]; [4; 1] ]
/* on rajoute hd partout */
r = List.map (fun v -> 3::v) r = [ [3; 1; 4] ; [3; 4; 1] ]
/* on ajoute le cas ou x est en tete de l */
r = (1::[3;4])::r
/* resultat */
r = [ [1; 3; 4]; [3; 1; 4]; [3; 4; 1] ]
```

```
let rec insert x l =
  match l with
  | [] -> [ [x] ]
  | hd :: tl ->
    (* on insere x dans tl *)
    let r = insert x tl in
    (* on rajoute hd partout *)
    let r = List.map (fun v -> hd::v) r in
    (* on ajoute le cas ou x est devant l *)
    (x :: l) :: r
```

Exercice 2 Avec `List.fold_left`, écrire `permutations : 'a list -> 'a list list` tq `permutations l` renvoie la liste des permutations de la liste `l`.

Exemple `permutations [1;2;3] ->`
`[ [1; 3; 2]; [3; 1; 2]; [3; 2; 1]; [1; 2; 3]; [2; 1; 3]; [2; 3; 1] ]`

Solution Idée de l'algorithme pour calculer les permutations de l :

— si la liste est vide ou ne contient un seul élément, on a une seule permutation possible : `[l]`.

— sinon on doit calculer les permutations de hd : :tl : on calcule les permutations de tl, puis on parcourt la liste des permutations et pour chaque permutation on insere x a tous les endroits possibles. Par exemple :

```
/* cas ou l = [1;2;3], hd = 1, tl = [2;3] */
permutations [1;2;3]
/* calcul des permutations de tl */
r = permutations tl = [ [2; 3]; [3; 2] ]
/* pour chaque element de tl, on insere hd */
/* et on concatene la liste obtenue a un accumulateur */
r = List.fold_left (fun acc x -> (insert hd x)@acc) [] r
  = (insert 1 [3; 2])@(insert 1 [2; 3])@[]
  = [ [1;3;2]; [3;1;2]; [3;2;1] ]@[ [1;2;3]; [2;1;3]; [2;3;1] ]@[]
  = [ [1; 3; 2]; [3; 1; 2]; [3; 2; 1]; [1; 2; 3]; [2; 1; 3]; [2; 3; 1]]
```

```ocaml
let rec permutations l =
  match l with
  | [] | [_] -> [ l ]
  | hd::tl ->
    (* calcul des permutations de tl *)
    let r = permutations tl in
    (* pour chaque element de tl, on insere hd
     * et on concatene la liste obtenue a un accumulateur *)
    List.fold_left (fun acc p -> (insere hd p) @ acc) [] r
```

2 Arbres binaires

```ocaml
type 'a t = V | N of 'a t * 'a * 'a t
```

Exercice 3 Etant donné un type `type p = Pre | Inf | Post`, écrire une fonction `print : p -> int t -> unit` tq `print p` affiche l'arbre en mode préfixe, infixe, ou postfixe.

Exemple En préfixe on regarde la racine, le sous-arbre gauche puis le sous-arbre droit, en infixe on regarde le sous-argre gauche, la racine, puis le sous-arbre droit et en postfixe on regarde les deux sous-arbres puis la racine.

— Parcours de l'arbre (fig 1) en préfixe : 3 2 1 4 5.
— Parcours de l'arbre (fig 1) en infixe : 1 2 3 4 5.
— Parcours de l'arbre (fig 1) en postfixe : 1 2 5 4 3.

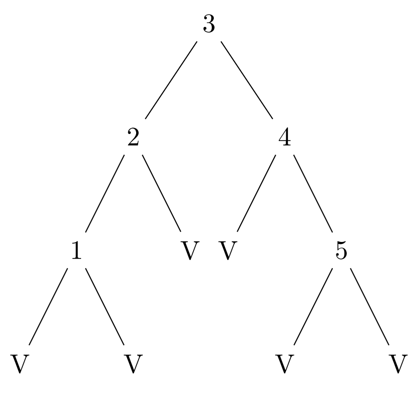

Figure 1 – Exemple d'arbre binaire

On présente ici plusieurs façons de faire. Les deux premières ont été vues en TD, la troisième est encore plus concise. NB : elles font toutes le même parcours pour un arbre donné, il s'agit surtout du nombre de lignes à écrire et de dupliquer le moins possible le code.

Solution 1

```
let rec print_pre a =
  match a with
  | V -> ()
  | N(g, x, d) -> print_int x ; print_pre g ; print_pre d

let rec print_inf a =
  match a with
  | V -> ()
  | N(g, x, d) -> print_inf g ; print_int x ; print_inf d

let rec print_post a =
  match a with
  | V -> ()
  | N(g, x, d) -> print_post g ; print_post d ; print_int x

let print p a =
  match p with
  | Pre -> print_pre a
  | Inf -> print_inf a
  | Post -> print_post a
```

Solution 2

```
let rec print p t =
  match t with
  | V -> ()
  | N(g, x, d) ->
    match p with
    | Pre -> print_int x ; print p g ; print p d
    | Inf -> print p g ; print_int x ; print p d
    | Post -> print p g ; print p d ; print_int x
```

Solution 3

```
let rec print p a =
  match a with
  | V -> ()
  | N(g, x, d) ->
    if p = Pre then print_int x;
    print p g;
    if p = Inf then print_int x;
    print p d;
    if p = Post then print_int x;
```

Exercice 4 Ecrire `min_max : 'a t -> 'a * 'a` tq `min_max a` renvoie le minimum et le maximum des valeurs contenues dans a, si a est non vide, en le parcourant une seule fois.

Exemple Pour l'arbre de la figure 1, le minimum vaut 1 et le maximum 5.

Solution Idée de l'algorithme pour obtenir le min et le max d'un arbre a :

- si a est vide, on a une erreur
- si a est un noeud avec deux sous-arbres vides et une racine r, le minimum et le maximum sont r
- si a est un noeud avec un sous-arbre non vide sa, un sous-arbre vide et une racide n, on calcule recursivement le minimum et le maximum de st et on compare à x
- si a est un noeud avec deux sous-arbres vides g et d et une racine r, on calcule recursivement les minima et maxima de g et d et on les compare entre eux, et à x

```
let rec min_max t =
  match t with
  (* on a suppposé que le cas "arbre vide" ne peut pas arriver:
   * on ajoute donc une assertion qui sera toujours fausse *)
  | V -> assert false
  | N(V, x, V) -> x, x
  | N(sa, x, V) | N(V, x, sa) ->
    let inf, sup = min_max sa in
    min inf x, max sup x
  | N(g, x, d) ->
    let inf_g, sup_g = min_max g in
    let inf_d, sup_d = min_max d in
    min (min inf_g inf_d) x, max (max inf_g inf_d) x
```

NB : on ne peut pas faire facilement de récursif terminal parce qu'il nous faut les résultats des appels récursifs sur les deux sous-arbres. Il faudrait utiliser du CPS.

3 Arbres n-aires de syntaxe abstraite

```
type exp = C of int | V of string | Plus of exp list | Mult of exp list
```

Exercice 5 Représenter $(2 + x + 5) \times 2 \times (1 + (-2 \times y) + -3)$.

```
let expr =
  Mult [ Plus [ C 2; V "x"; C 5 ];
         C 2;
         Plus [ C 1; Mult [ C (-2); V "y" ]; C (-3) ]
       ]
```

```
type env = (string * int) list
```

Exercice 6 Ecrire `eval : expr -> env -> int` tq `eval e [(x1, v1); ...; (xn, vn)]` évalue e en considérant que xi = vi.

Exemple

```
  eval expr [("x", 3); ("y", 0)]
= (2 + env("x") + 5) * 2 * (1 + (-2 * env("y")) + -3)
= (2 + 3 + 5) * 2 * (1 + (-2 * 0) + -3)
= 10 * 2 * (-2)
= -40
```

Solution Idée de l'algorithme pour évaluer une expression e :
- si e est une constante de valeur n, on renvoie n
- si e est une variable de valeur x, on renvoie la valeur associée à x dans l'environnement
- si e est une addition sur l, on parcourt l et pour chaque élément on évalue sa valeur et on ajoute le résultat à un accumulateur
- si e est une multiplication sur l, idem sauf qu'on multiplie

```
let rec eval env e =
  match e with
  | C n -> n
  | V x -> List.assoc x env
  | Plus l -> List.fold_left (fun acc v -> acc + eval env v) 0 l
  | Mult l -> List.fold_left (fun acc v -> acc * eval env v) 1 l
```

NB : pour la multiplication il faut initialiser à 1, l'élément neutre de la multiplication.