Révisions et automatismes

OPTION INFORMATIQUE - TP n° 3.0 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ implémenter les opérations élémentaires récursives sur les listes
- ☞ utiliser les tableaux et la programmation impérative
- ☞ implémenter le tri par insertion, le tri fusion et le tri rapide
- ☞ implémenter les opérations élémentaires récursives sur les arbres binaires
- ☞ effectuer une démonstration par récurrence
- ☞ effectuer une démonstration par induction structurelle

A Listes et récursivité

Écrire des fonctions OCaml qui implémentent les opérations suivantes sur les listes :

A1. Calculer la longueur d'une liste (`length`).

A2. Déterminer le i-ème élément d'une liste (`at` ou `nth`). On considèrera deux solutions : la première échoue en lançant une exception si l'élément n'est pas trouvé. La seconde renvoie un type `option`.

A3. Tester l'appartenance d'un élément à une liste (`mem`).

A4. Concaténer de deux listes (`concat`).

A5. Aplatir une liste de listes (`flatten`).

A6. Appliquer une fonction à tous les éléments d'une liste et retourner la liste correspondante : essayer de le faire en temps linéaire (`map`).

A7. Extraire les deux derniers éléments d'une liste. Cette fonction renvoie une tuple. Fonctionne-t-elle sur une liste de liste ?

A8. Construire la liste inverse d'une liste, c'est à dire que l'élément 0 devient le dernier élément, le premier l'avant dernier…

Solution :

Code 1 – Listes et récursivité

```
let test_list = [1;42;99;66;12;7]
let test_list_2 = [0;-3;-23]
let test_list_list = [[0;1;2]; [3;4]; [5;6;7;8];[9];[10;11]]

(* not tail recursive *)
let rec length l =
```

```
  match l with
    | [] -> 0
    | h::t -> 1 + length t;;
length test_list;;

(* tail recursive *)
let length l =
    let rec aux n = function
      | [] -> n
      | _::t -> aux (n + 1) t
    in
    aux 0 l;;
length test_list;;

(* raise exception if not found *)
let rec at k l =
  match l with
    | [] -> failwith "not found !"
    | h::t when k = 0 -> h
    | h::t -> at (k - 1) t ;;
at 3 test_list;;

(* return an option *)
let rec at k l = match l with
    | [] -> None
    | h :: t -> if k = 0 then Some h else at (k - 1) t;;
at 3 test_list;;

let rec mem x l =
  match l with
    | [] -> false
    | h::t when h = x -> true
    | h::t -> mem x t;;
mem 66 test_list;;
mem 0 test_list;;


let rec concat l1 l2 =
  match l1 with
    | [] -> l2
    | a::q -> a::(concat q l2) ;;
concat test_list test_list_2;;


let rec flatten l =
  match l with
    | [] -> []
    | h::t -> concat h (flatten t) ;;
flatten test_list_list;;

(* @ operator *)
let rec flatten l =
  match l with
    | [] -> []
    | h::t ->  h @ (flatten t) ;;
```

```
flatten test_list_list;;

let rec iter f l =
  match l with
    | [] -> []
    | h::t -> f(h); iter f t;;
iter (fun x -> print_int (x*x); print_newline()) test_list;;

let rec map f l =
  match l with
    | [] -> []
    | h::t -> f(h)::(map f t) ;;
map (fun x -> x*x) test_list;;

let rec last_two l =
    match l with
        | [] | [_] -> failwith "not enough elements !"
        | [a; b] -> (a,b)
        | _::t -> last_two t;;
last_two test_list;;
last_two test_list_list;;

let rev list =
    let rec aux built l =
        match l with
          | [] -> built
          | h::t -> aux (h::built) t
    in
    aux [] list;;
rev test_list;;
rev test_list_list;;
```

B Vecteurs et programmation impérative

B1. Écrire une fonction qui calcule le n-ième terme de la suite de Fibonacci en utilisant un vecteur où sont stockées les valeurs successives. Quelle est sa complexité temporelle ? spatiale ?

B2. Écrire une fonction qui calcule le n-ième terme de la suite de Fibonacci en utilisant des références. Quelle est sa complexité temporelle ? spatiale ?

B3. Écrire une fonction `mem x tab` qui test l'appartenance d'un élément `x` à un vecteur `tab`. Quelle est sa complexité ? Peut-on faire mieux si l'on suppose le tableau trié ? Programmer une version efficace dans ce cas.

B4. Écrire une fonction `v_concat v1 v2` qui concatène deux tableaux unidimensionnels et renvoie le nouveau tableau ainsi créé.

B5. Écrire une fonction `m_concat m1 m2` qui concatène deux matrices ayant le même nombre de lignes.

B6. Écrire une fonction `array_map f tab` qui, sur le modèle de la fonction `map` pour les listes, renvoie un nouveau vecteur contenant les images des éléments de `tab` par la fonction `f`.

Solution :

Code 2 – Vecteurs

```
(* Suite de Fibonacci - Array *)

let fibo n =
  let tab = Array.make (n + 1) 1 in
  for i = 2 to n do
    tab.(i) <- tab.(i - 1) + tab.(i - 2)
  done;
  tab.(n) ;;
fibo 6;;

(* Suite de Fibonacci - references *)

let fib n =
  let u0 = ref 1 and u1 = ref 1  in
  for i = 2 to n do
    let tmp = !u0 in  (* ne pas écraser une variable !*)
        u0 := !u1;
        u1 := tmp + !u1 ;
  done;
  !u1 ;;
fib 6;;


let test_tab = [|13;66;1;42;17;9;33;11|];;
let test_sorted_tab = [|1;9;11;13;17;33;42;66|];;


let mem x tab =
  let n = Array.length tab and b = ref false and i = ref 0 in
  while (not !b) && (!i < n) do
    if tab.(!i) = x then
      b := true;
    incr(i) ;
  done;
  !b ;;
mem 13 test_tab;;
mem 2 test_tab;;



(* Si tab est trié, recherche dichotomique *)

let dicho_mem x tab =
  let n = Array.length tab and b = ref false  in
  let g = ref 0 and d = ref (n - 1) in  (* indices de début et de fin *)
  while (not !b) && g <= d do
    let m = ((!g) + (!d)) / 2 in
    if tab.(m) = x then
      b := true
    else if tab.(m) < x then
      g := m + 1
    else
```

```
      d := m - 1
  done ;
  !b ;;
dicho_mem 13 test_sorted_tab;;
dicho_mem 2 test_sorted_tab;;

(* version récursive *)
let rec_dicho_mem x tab =
  let rec aux g d =
    if g > d
    then false
    else let m = (g+d)/2 in
         if tab.(m) = x
             then true
         else if tab.(m) < x
             then aux (m+1) d
         else aux g (m-1)
  in aux 0 (Array.length tab - 1) ;;
rec_dicho_mem 13 test_sorted_tab;;
rec_dicho_mem 2 test_sorted_tab;;

(* Concaténation de vecteurs *)

let v_concat v1 v2 =
  let n1 = Array.length v1 and n2 = Array.length v2 in
  let a = Array.make (n1 + n2) v1.(0) in
  for i = 1 to n1 - 1 do
    a.(i) <- v1.(i)
  done;
  for i = 0 to n2 - 1 do
    a.(n1 + i) <- v2.(i)
  done;
  a ;;
v_concat test_tab test_sorted_tab;;

(* Concaténation de matrices *)

let m_concat m1 m2 =
  let nl1 = Array.length m1 and nl2 = Array.length m2 in
  if nl1 <> nl2 then
    failwith "wrong size : can not concat"
  else
    let m = Array.make nl1 [| |] in
    for i = 0 to nl1-1 do
      m.(i) <- v_concat m1.(i) m2.(i)
    done;
  m ;;
let m1 = [|[|13;66|];[|1;42;17|];[|9;33;11|]|];;
let m2 =  [|[|3;6|];[|0;41;16|];[|8;32;10|]|];;
let m3 =  [|[|0;41;16|];[|8;32;10|]|];;
m_concat m1 m2;;
m_concat m1 m3;;

(* map pour les array *)
let array_map f tab =
```

```
    let n = Array.length tab in
    let out = Array.make n 0 in
    for i = 0 to n-1 do
      out.(i) <- f(tab.(i))
    done;
    out ;;
  array_map (fun x -> x*x) test_tab;;
```

C Tris

C1. Écrire une fonction qui effectue le tri par insertion. On décompose le problème comme suit :

(a) Écrire une fonction récursive de protoype `val insert_elem : 'a list -> 'a -> 'a list` qui insère un élément dans une liste triée à la bonne place.

(b) Écrire une fonction de prototype `val insert_sort : 'a list -> 'a list` qui trie par insertion une liste.

(c) Utiliser les fonctions `List.fold_left` et `insert_elem` pour écrire une fonction le tri par insertion en une seule ligne.

C2. Discuter la terminaison, la correction et la complexité de la fonction `insert_elem`.

Solution :

- Terminaison : la fonction `insert_elem` effectue un appel récursif sur une liste dont la taille est strictement plus petite, car on a enlevé un élément : cette taille est une suite positive strictement décroissante (entière, positive, strictement décroissante) et atteint nécessairement zéro. Deux cas sont possibles alors : soit la fonction est appelée sur une liste vide et termine, soit la fonction extrait un élément en tête supérieur à l'élément à insérer et termine. Dans les deux cas, la fonction se termine.
- Correction : on procède par récurrence sur la taille de la liste.
  Soit $\mathcal{P}_n$ la propriété suivante : «pour tout liste `sorted` triée de taille n, `insert_elem sorted e` renvoie une liste triée dont les éléments sont ceux de `sorted` et `e`».
  1. Initialisation : $\mathcal{P}_0$ est vraie car un singleton `[e]` est trié.
  2. Hérédité : Supposons que $\mathcal{P}_n$ soit vraie. Soit une liste `l = h::t` de taille $n+1$ et triée. Si l'élément à insérer `e` est plus petit que `h`, alors il est inséré en tête et la liste `e::h::t` est correctement triée, puisque par hypothèse de `h::t` est triée. Si l'élément `e` à insérer est plus grand que `h`, alors `(insert_elem t e)` est une liste triée puisque `t` est de taille n et que $\mathcal{P}_n$ est vraie. `h::(insert_elem t e)` est donc une liste triée. Donc, si $\mathcal{P}_n$ est vraie, $\mathcal{P}_{n+1}$ est vraie.
  3. Conclusion : $\mathcal{P}_n$ est vrai pour tout n.

C3. Discuter la terminaison, la correction et la complexité de la fonction `insert_sort`.

Solution :

- Terminaison : la fonction `insert_sort` effectue un appel récursif sur une liste dont la taille est strictement plus petite, car on a enlevé un élément (la taille est une suite positive strictement décroissante). Ainsi, l'appel aboutit nécessairement sur une liste vide et la fonction termine.
- Correction : on procède par récurrence sur la taille de la liste.
  Soit $\mathcal{P}_n$ la propriété suivante : «pour une liste `to_sort` de taille n et une liste `sorted` triée, `aux to_sort sorted` renvoie une liste triée dont les éléments sont ceux de `to_sort` et `sorted`».
  1. Initialisation : $\mathcal{P}_0$ est vraie car `sorted` est triée et le singleton `[]` ne contient aucun élément.
  2. Hérédité : Supposons que $\mathcal{P}_n$ soit vraie. Soit une liste `l = h::t` de taille $n+1$. La fonction effectue l'appel récursif `aux t (insert_elem sorted e)`. Or, l'insertion de l'élément est correct, donc le second paramètre `(insert_elem sorted e)` est une liste bien triée. Le premier paramètre est de taille n. Par hypothèse de récurrence, cet appel récursif renvoie donc bien une liste triée. Donc, si $\mathcal{P}_n$ est vraie, $\mathcal{P}_{n+1}$ est vraie.
  3. Conclusion : Comme $\mathcal{P}_0$ est vraie et que l'hérédité est démontrée, $\mathcal{P}_n$ est vrai pour tout n.
- Complexité : dans le pire des cas, le tableau est trié dans l'ordre inverse et tous les éléments vont parcourir l'intégralité du tableau de gauche lors de l'insertion : la complexité est alors en $O(n^2)$ puisque `insert_elem` est en $O(n)$ et qu'on parcourt tous les éléments de la liste dans `insert_sort`. Dans le meilleur des cas, le tableau est trié et la fonction `insert_elem` est à coût constant $O(1)$ puisque l'élément est déjà inséré à la bonne place. La complexité vaut alors $O(n)$.

C4. Écrire une fonction qui réalise le tri fusion. On décompose le problème comme suit :
   (a) une fonction qui sépare une liste en deux,
   (b) une fonction qui réalise la fusion de deux listes triées en une seule liste triée,
   (c) une fonction qui effectue le tri fusion.

C5. Discuter la terminaison, la correction et la complexité du tri fusion.

Solution :

1. Terminaison : les appels récursifs se font sur des listes dont la taille est strictement plus petite. Ils aboutissent donc nécessairement sur une liste vide, la condition d'arrêt est atteinte et la fonction se termine.
2. Correction : Soit $\mathcal{P}_n$ la propriété suivante : «pour toute liste `l` de taille inférieure ou égale à n, `merge_sort l` renvoie une liste triée dont les éléments sont ceux de `l`».
   (a) Initialisation : $\mathcal{P}_0$ est vraie car le singleton `[]` est trié et ne contient aucun élément.
   (b) Hérédité : Supposons que $\mathcal{P}_n$ soit vraie. Soit une liste `l = h::t` de taille $n+1$. La fonction effectue l'appel récursif `merge (merge_sort l1)(merge_sort l2)`. D'après l'hypothèse de récurrence, comme les tailles de `l1` et `l2` sont inférieures à n puisque créées à partir de `l` qui est de taille $n+1$, les deuxième et troisième paramètres sont

des listes bien triées. Si on suppose la fusion des deux listes correctes, alors cet appel récursif renvoie donc bien une liste triée. Donc, si $\mathcal{P}_n$ est vraie, $\mathcal{P}_{n+1}$ est vraie.

(c) Conclusion : $\mathcal{P}_n$ est vrai pour tout n.

3. Complexité : on a la récurrence suivante $T(n) = 2T(n/2) + cn$, car la complexité de la fonction `merge` est en $O(n)$. La complexité du tri fusion est donc de en $O(n\log(n))$ (dans tous les cas).

C6. Écrire une fonction qui réalise le tri rapide. On décompose le problème comme suit :

(a) une fonction qui partitionne une liste en deux d'après le pivot,

(b) une fonction qui effectue le tri rapide.

C7. Discuter la terminaison, la correction et la complexité du tri rapide.

Solution : Le pire des cas pour le tri rapide est lorsque le pivot est le plus grand ou le plus petit, et ce à chaque fois qu'on le choisit. Dans ce cas, les deux partitions sont complètement déséquilibrées et la complexité est en $O(n^2)$. Ce revient à faire un tri bulle bidirectionnel.

Code 3 – Tris

```
let test_sorted_list = [1;7;12;27;36;42;54;66;73;88;99]
let test_list = [1;42;99;66;36;12;7;27;73;36;54;88]

(* tri par insertion *)

let rec insert_elem sorted e =
    match sorted with
        | [] -> [e]
        | h::t -> if h < e then h::(insert_elem t e) else e::h::t;;
test_list;;
insert_elem test_sorted_list 5;;

let insert_sort l =
    let rec aux to_sort sorted =
        match to_sort with
           | [] ->  sorted
           | e::t -> aux t (insert_elem sorted e)
    in aux l [];;
insert_sort test_list;;

let fold_left_insert_sort l = List.fold_left insert_elem [] l;;
test_list;;
fold_left_insert_sort test_list;;



(* tri fusion *)

let rec slice l =
  match l with
    | [] -> ([],[])
    | [a] -> ([a],[])
    | a::b::t -> let (l1,l2) = slice t in (a::l1, b::l2);;
```

```
let rec merge l1 l2 =
  match (l1,l2) with
    | ([],l2) -> l2
    | (l1,[]) -> l1
    | (a1::t1, a2::_) when a1 < a2 -> a1::(merge t1 l2)
    | (_::_, a2::t2) -> a2::(merge l1 t2) ;;

let rec merge_sort l =
  match l with
    | [] -> []
    | [a] -> [a]
    | l -> let (l1,l2) = slice l
           in merge (merge_sort l1) (merge_sort l2) ;;
test_list;;
merge_sort test_list;;


(* tri rapide *)
let rec partition l pivot=
    match l with
    | [] -> [],[]
    | t::q -> let (l1,l2) = partition q pivot in
                    if (t < pivot) then (t::l1,l2) else (l1,t::l2);;

let rec quick_sort l =
    match l with
    | []->[]
    | h::t ->
        let (l1,l2) = partition t h in
            (quick_sort l1)@(h::(quick_sort l2));;
test_list;;
quick_sort test_list;;
```

D Arbres binaires de recherche

On considère la définition récursive du type `'a arbre` :

```
type 'a tree = Nil | Node of 'a tree * 'a * 'a tree ;;
```

Écrire les fonctions qui implémentent les opérations suivantes sur un type `tree` :

D1. Calculer la hauteur d'un arbre binaire.

D2. Calculer le nombre total de nœuds d'un arbre binaire.

D3. Calculer le nombre de feuilles d'un arbre binaire.

D4. Calculer récursivement le nombre de nœuds internes d'un arbre binaire. Démontrer par induction structurelle que cette fonction est correcte.

Solution : Soit $\mathcal{P}(t)$: «La fonction `internal_nodes_nb` appliquée à l'arbre t renvoie le nombre de nœuds internes de l'arbre». La démonstration par induction structurelle procède comme suit :

1. (**CAS DE BASE**) Tout d'abord, l'arbre vide `Nil`. La fonction renvoie 0 qui est le nombre de nœuds (internes) de l'arbre.
2. (**PAS D'INDUCTION**) Soit $e \in E$ une étiquette et $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ deux arbres binaires pour lesquels $\mathcal{P}(f_g)$ et $\mathcal{P}(f_d)$ sont vraies : la fonction `internal_nodes_nb` renvoie le nombre de nœuds internes de ces arbres. Pour l'arbre construit selon la règle de la structure, c'est-à-dire (f_g, e, f_d), la fonction renvoie un de plus que le nombre de nœuds des fils. Ce qui est exact puisqu'on a ajouté un nœud à l'arbre. Donc, $\mathcal{P}\big((f_g, e, f_d)\big)$ est vraie.
3. (**CONCLUSION**) Quelque soit l'arbre t, $\mathcal{P}(t)$ est vraie.

D5. Ajouter un élément à la fin de la branche la plus à gauche et renvoyer le nouvel arbre.

Solution :

Code 4 – Arbres binaires

```
type 'a tree = Nil | Node of 'a tree * 'a * 'a tree ;;

let t = Node(
    Node(Node(Nil, 3, Nil), 7, Node( Nil, 9, Nil)),
    13,
    Node(Node(Nil, 4, Node(Nil, 2, Nil)), 17, Node(Nil, 7, Nil))
  );;

let rec height a =
  match a with
    | Nil -> -1
    | Node(fg,_,fd) -> 1 + max (height fg) (height fd) ;;
height t;;

let rec size a =
  match a with
    | Nil -> 0
    | Node(fg, x, fd) -> 1 + size fg + size fd ;;
size t;;

let rec leaves_nb a =
  match a with
    | Nil -> 0
    | Node(Nil, x, Nil) -> 1
    | Node(fg, x, fd) -> leaves_nb fg + leaves_nb fd ;;
leaves_nb t;;

let internal_nodes_nb a = size a - leaves_nb a ;;
internal_nodes_nb t;;

let rec internal_nodes_nb a =
  match a with
    | Nil -> 0
    | Node(Nil,_,Nil) -> 0
```

```
    | Node(fg,x,fd) -> 1 + internal_nodes_nb fg + internal_nodes_nb fd ;;
 internal_nodes_nb t;;


let rec add e a =
  match a with
    | Nil -> Node(Nil,e,Nil)
    | Node(fg,x,fd) -> Node((add e fg), x, fd) ;;
add 23 t;;
```