Correction du DS 2

Option informatique, première année

Julien Reichert

Exercice 1

```
(* minimum : 'a vect -> 'a *)
let minimum tab =
  let n = vect_length tab in
    if n = 0 then failwith "Tableau vide"
    else let mini = ref tab.(0) in
      for i = 1 to n-1 do
        if tab.(i) < !mini
          then mini := tab.(i)
      done; !mini;;
```

Exercice 2

```
(* minimum_liste : 'a list -> 'a *)
let rec minimum_liste = function
|[] -> failwith "Liste vide"
|[a] -> a
|a::q -> min a (minimum_liste q);;

(* Version récursive terminale : *)
let minimum_liste l =
  let rec minaux mini = function
  | [] -> mini
  | a::q -> minaux (min a mini) q
  in minaux (hd l) (tl l);;
(* Si liste vide, Failure "hd" *)
```

Exercice 3

Question 3.0

Il est évident que plus on a de nombres positifs, plus la somme est grande. Ainsi, toute zone incomplète peut être complétée en se décalant du bord, sans perdre d'indice mais en en ajoutant, pour une somme au moins aussi bonne.

Question 3.1

Attention, on a dit qu'il y avait des flottants, donc on écrit `+.` au lieu de `+` (dans tout l'exercice).

```
let indice_max_somme tab =
  match vect_length tab with
  | 0 -> failwith "Tableau vide"
  | 1 -> 0 (* arbitraire *)
  | 2 -> 0 (* idem *)
  | 3 -> 1 (* évident, mais on pourrait fusionner avec le cas suivant *)
  | _ ->
    let ind_max = ref 1 and somme_max = ref (tab.(0) +. tab.(1) +. tab.(2)) in
      for i = 2 to vect_length tab - 2 do
        let somme = tab.(i-1) +. tab.(i) +. tab.(i+1) in
          if somme > !somme_max
            then begin somme_max := somme; ind_max := i end
      done; !ind_max;;
```

Question 3.2

```
let somme_carre mat i j =
  mat.(i-1).(j-1) +. mat.(i-1).(j) +. mat.(i-1).(j+1)
  +. mat.(i).(j-1) +. mat.(i).(j) +. mat.(i).(j+1)
  +. mat.(i+1).(j-1) +. mat.(i+1).(j) +. mat.(i+1).(j+1);;
(* On est à la limite de faire une boucle. *)
```

Question 3.3

```
(* Programme naïf qu'on optimisera dans la dernière question. *)
let plus_grande_somme_carre_bonne_taille mat =
  let ligne_max = ref 1 and colonne_max = ref 1 and somme_max = ref (somme_carre mat 1 1) in
  for i = 1 to vect_length mat - 2 do
    (* On suppose la matrice carrée, sinon ça peut faire mal. *)
    for j = 1 to vect_length mat.(i) - 2 do
      let somme = somme_carre mat i j in
        if somme > !somme_max
          then begin somme_max := somme; ligne_max := i; colonne_max := j end
    done
  done; (!ligne_max, !colonne_max);;

let plus_grande_somme_carre mat =
  let lignes = vect_length mat in
    if lignes = 0 then failwith "Matrice vide";
  let colonnes = vect_length mat.(0) in
    if colonnes = 0 then failwith "Matrice vide";
  if lignes = 1
    then let ind = indice_max_somme mat.(0) in (0,ind)
  else if lignes = 2
    then let ind = indice_max_somme (init_vect colonnes (fun i -> mat.(0).(i) +. mat.(1).(i))) in (0,ind)
  else if colonnes = 1
    then let ind = indice_max_somme (init_vect colonnes (fun i -> mat.(i).(0))) in (ind,0)
  else if colonnes = 2
    then let ind = indice_max_somme (init_vect colonnes (fun i -> mat.(i).(0) +. mat.(i).(1))) in (ind,0)
  else plus_grande_somme_carre_bonne_taille mat;;
(* Traitement un peu bourrin des tailles pathologiques. *)
```

Question 3.4

Puisqu'on aura un grand nombre de récupérations de valeurs pour la fonction `somme_carre`, autant l'appeler une fois pour toutes avec toutes les positions possibles, et stocker les résultats dans une variable globale accessible à tout moment pour ne faire qu'une consultation au lieu de neuf calculs à chaque fois que le besoin s'en fait sentir. Un raffinement serait même de ne mettre à jour la variable globale que si nécessaire et donc de calculer chaque valeurs au premier appel de la position correspondante.

Question 3.5

```
let mat = make_matrix 100 100 0.;;
for i = 0 to 99 do for j = 0 to 99 do mat.(i).(j) <- random__float 100. done done;;

let mat_appui = make_matrix (vect_length mat) (vect_length mat.(0)) (-.1.);;

let somme_carre i j = if mat_appui.(i).(j) > -.0.5 then mat_appui.(i).(j)
  else
    let hg = (if i > 0 && j > 0 then mat.(i-1).(j-1) else 0.)
    and h = (if i > 0 then mat.(i-1).(j) else 0.)
    and hd = (if i > 0 && j < vect_length mat.(0) - 1 then mat.(i-1).(j+1) else 0.)
    and g = (if j > 0 then mat.(i).(j-1) else 0.)
    and d = (if j < vect_length mat.(0) - 1 then mat.(i).(j+1) else 0.)
    and bg = (if i < vect_length mat - 1 && j > 0 then mat.(i+1).(j-1) else 0.)
    and b = (if i < vect_length mat - 1 then mat.(i+1).(j) else 0.)
    and bd = (if i < vect_length mat - 1 && j < vect_length mat.(0) - 1 then mat.(i+1).(j+1) else 0.) in
      let somme = hg +. h +. hd +. g +. mat.(i).(j) +. d +. bg +. b +. bd in
        mat_appui.(i).(j) <- somme; somme;;
```

Si on veut remplir complètement la matrice d'appui, on peut écrire le programme suivant. Pour le plaisir, le remplissage est fait en « boustrophédon », de sorte qu'au lieu de neuf additions par position, on se limite à trois additions et trois soustractions. Oui, c'est trop la classe.

```
(* On suppose que la matrice a au moins trois lignes et trois colonnes, sinon -> dimension 1. *)
let remplit_mat_appui () =
  let lignes = vect_length mat and colonnes = vect_length mat.(0) and l = ref 0 and c = ref 0 in
    let somme_actuelle = ref (mat.(0).(0) +. mat.(1).(0)) in
      while !l < lignes do
        let signe = (if !l mod 2 = 0 then 1. else -1.) in let signe_int = int_of_float signe in
          while !c >= 0 && !c < colonnes do
            if !c-2*signe_int >= 0 && !c-2*signe_int < colonnes
              then for i = max 0 (!l-1) to min (lignes-1) (!l+1)
                do somme_actuelle := !somme_actuelle -. mat.(i).(!c-2*signe_int) done;
            if !c+signe_int < colonnes && !c+signe_int >= 0
              then for i = max 0 (!l-1) to min (lignes-1) (!l+1)
                do somme_actuelle := !somme_actuelle +. mat.(i).(!c+signe_int) done;
            mat_appui.(!l).(!c) <- !somme_actuelle;
            c := !c + signe_int
          done;
          c := !c - signe_int; (* Retour à une position autorisée *)
        if !l > 0 && !l < lignes - 1 then for j = max 0 (!c-1) to min (colonnes-1) (!c+1)
          do somme_actuelle := !somme_actuelle -. mat.(!l-1).(j) done;
        if !l < lignes - 2 then for j = max 0 (!c-1) to min (colonnes-1) (!c+1)
          do somme_actuelle := !somme_actuelle +. mat.(!l+2).(j) done;
        incr l;
        if !l < lignes then mat_appui.(!l).(!c) <- !somme_actuelle; c := !c - signe_int
      done;;
```

Bien entendu, l'optimisation ne mérite pas le mal de tête, mais écrire un programme qui fait la même chose que `somme_carre` dans une double boucle avait l'air trop commun...

Question 3.6

Le traitement bourrin des cas pathologiques resterait valable, mais il faudrait le modifier. Mieux vaut changer la première fonction pour adapter la zone.

```
let somme_zone mat taille i j =
  let lignes = vect_length mat and colonnes = vect_length mat.(0) and somme = ref 0. in
    for l = max 0 (i - taille) to min (lignes - 1) (i + taille) do
      for c = max 0 (j - taille) to min (colonnes - 1) (j + taille) do
        somme := !somme +. mat.(l).(c)
      done
    done; !somme;;

let plus_grande_somme_zone mat taille =
  let lignes = vect_length mat and colonnes = vect_length mat.(0) in
    if lignes = 0 || colonnes = 0 then failwith "Matrice vide";
    let ligne_max = ref taille and colonne_max = ref taille
    and somme_max = ref (somme_zone mat taille taille taille) in
      for i = taille to vect_length mat - taille - 1 do
        for j = taille to vect_length mat.(i) - taille - 1 do
          let somme = somme_zone mat taille i j in
            if somme > !somme_max
              then begin somme_max := somme; ligne_max := i; colonne_max := j end
        done
      done; (!ligne_max, !colonne_max);;
(* On pourrait ramener à lignes/2 ou colonnes/2 pour centrer si la matrice est trop petite. *)
```

Exercice 4

Question 4.0

Un arbre binaire est soit vide soit une feuille soit un nœud ayant deux fils, qui sont de manière récursive deux arbres binaires. Si on veut stocker l'information portée par une feuille dans une case d'un tableau, il faut prendre un constructeur qui le précise, car les éléments d'un tableau doivent tous être de même type. De plus, les noms de constructeurs doivent toujours être différents d'un type à l'autre. Enfin, dans notre tableau, un nœud est la donnée de son information ainsi que des indices des positions des fils, éventuellement vides, du nœud. L'ordre était ici arbitraire.

Question 4.1

D'après le cours, un arbre binaire ayant n feuilles a $n-1$ nœuds internes. Ainsi, on écrit :

```
let initialise_arbre_bin_tab n = make_vect (2*n-1) Rien;;
```

On note que le doute demeure quant aux types des informations sur les nœuds et les feuilles, ce qui se remarque à la signature de la fonction.

Question 4.2

La structure de données récursive laisse peu de place au doute quant à la nature de la fonction...

```
let rec taille = function
| Vide -> 0
| Feuille(_) -> 1
| Noeud(g,_,d) -> 1 + taille g + taille d;;

let arbre_bin_to_vect a =
  let n = taille a in
    let tab = make_vect n Rien in
(* Pas la fonction précédente, on ne va pas transformer n pour refaire le calcul à l'envers ! *)
      let rec remplit indice = function
      | Vide -> indice (* Vide n'est pas censé exister dans un arbre non vide. *)
      | Feuille(f) -> tab.(indice) <- Pos_feuille(f); indice + 1
      | Noeud(g,n,d) -> let nouvel_indice = remplit (indice+1) g in
        tab.(indice) <- Pos_noeud(indice+1,n,nouvel_indice);
        remplit nouvel_indice d
      in let _ = remplit 0 a in tab;;
```

Question 4.3

L'ordre des arguments n'est pas imposé, tant que la signature annoncée est cohérente. Cependant, il est recommandé que ce soit le même dans la question suivante.

```
(* ajouter_feuille_tab : ('a,'b) arbre_bin_tab -> 'a -> int list -> unit *)
let ajouter_feuille_tab tab etiquette positions =
  let rec parcourir indice = function
  | [] -> if tab.(indice) = Rien
              then tab.(indice) <- Pos_feuille(etiquette)
              else failwith "Position occupée"
  | pos::q ->
    begin match tab.(indice) with
    | Pos_noeud(g,_,d) -> if pos = 0 then parcourir g q else parcourir d q
    | _ -> failwith "Branche trop courte"
(* On peut vérifier que pos = 1 et planter sinon, mais pour une fois on sera tolérant. *)
    end
  in parcourir 0 positions;;
```

Question 4.4

La difficulté réside dans la recherche d'une ou plusieurs positions libres du tableau pour faire pointer notre nouveau nœud. On va modulariser au maximum, ici.

```
let premieres_positions_libres tab =
  let n = vect_length tab in
    let liberte = make_vect n true in
      for i = 0 to n-1 do
        match tab.(i) with
        | Rien -> ()
        | Pos_feuille(_) -> liberte.(i) <- false
        | Pos_noeud(g,_,d) -> liberte.(i) <- false; liberte.(g) <- false; liberte.(d) <- false
      done; liberte;;

let deux_pos_libres liberte =
    let n = vect_length liberte and ind = ref 0 and un = ref (-1) and deux = ref(-1) in
      while !ind < n && !un = -1 do if liberte.(!ind) then (un := !ind; ind := !ind+1) done;
      while !ind < n && !deux = -1 do if liberte.(!ind) then (deux := !ind ; ind := !ind +1) done;
      if !deux = -1 then failwith "Plus de place" else (!un,!deux);;
```

```
(* ajouter_noeud_tab : ('a,'b) arbre_bin_tab -> 'b -> int list -> unit *)
let ajouter_noeud_tab tab etiquette positions =
  let rec parcourir indice = function
  | [] ->
    let contenu = tab.(indice) and (un,deux) = deux_pos_libres (premieres_positions_libres tab) in
      tab.(un) <- contenu; (* En remplaçant éventuellement Rien par Rien *)
      tab.(deux) <- Rien; (* En prévision de l'effacement éventuel d'éléments... *)
      tab.(indice) <- Pos_noeud(hd deuxpos,etiquette,hd (tl deuxpos))
  | pos::q ->
    begin match tab.(indice) with
    | Rien -> failwith "Branche trop courte"
    | Pos_feuille(_) -> failwith "Branche trop courte"
    | Pos_noeud(g,_,d) -> if pos = 0 then parcourir g q else parcourir d q
    end
  in parcourir 0 positions;;
```

Question 4.5

Cet exercice a l'air compliqué, mais il est plus facile que les précédents dans la mesure où il n'y a pas de piège.

```
let fusion_arbres etiquette gauche droite =
  let ng = vect_length gauche and nd = vect_length droite in
    let tab = make_vect (ng+nd+1) Rien in
      tab.(0) <- Pos_noeud(1,etiquette,ng+1);
      for i = 0 to ng-1 do
        match gauche.(i) with
        | Rien -> tab.(i+1) <- Rien
        | Pos_feuille(f) -> tab.(i+1) <- Pos_feuille(f)
        | Pos_noeud(g,n,d) -> tab.(i+1) <- Pos_noeud(g+1,n,d+1)
      done;
      for i = 0 to nd-1 do
        match droite.(i) with
        | Rien -> tab.(i+ng+1) <- Rien
        | Pos_feuille(f) -> tab.(i+ng+1) <- Pos_feuille(f)
        | Pos_noeud(g,n,d) -> tab.(i+ng+1) <- Pos_noeud(g+ng+1,n,d+ng+1)
      done;
      tab;;
```