```
type vecteur == (int*float) list;;
type matrice == (int*vecteur) list;;

(*********************** Question 3 ***********************)

let rec add plus null v1 v2 = match v1,v2 with
|  (n1,x1) :: r1, (n2,x2) :: r2 when n1 > n2 ->
    (n2,x2) :: (add plus null v1 r2)
| (n1,x1) :: r1, (n2,x2) :: r2 when n1 < n2 ->
    (n1,x1) :: (add plus null r1 v2)
| (n1,x1) :: r1, (n2,x2) :: r2 (* when n1 = n2 *)->
    let x = plus x1 x2 in
    if x <> null then
      (n1,(plus x1 x2)) :: (add plus null r1 r2)
    else
      add plus null r1 r2
| [], _ -> v2
| _, [] -> v1
;;

let (add_vect : vecteur -> vecteur -> vecteur) = add (prefix +.) 0. ;;
let (add_mat  : matrice -> matrice -> matrice) = add add_vect [] ;;

(*********************** Question 4 ***********************)

let rec scal v1 v2 = match v1,v2 with
| (n1,x1) :: r1, (n2,x2) :: r2 when n1 > n2 ->
    scal v1 r2
| (n1,x1) :: r1, (n2,x2) :: r2 when n1 < n2 ->
    scal r1 v2
| (n1,x1) :: r1, (n2,x2) :: r2 (* when n1 = n2 *) ->
    (x1 *. x2) +. (scal r1 r2)
| [], _ -> 0.
| _, [] -> 0.
 ;;

(*********************** Question 5 ***********************)

let rec add plus np cons nc v1 v2 = match v1,v2 with
|  (n1,x1) :: r1, (n2,x2) :: r2 when n1 > n2 ->
    cons (n2,(plus np x2)) (add plus np cons nc v1 r2)
| (n1,x1) :: r1, (n2,x2) :: r2 when n1 < n2 ->
    cons (n1,(plus x1 np)) (add plus np cons nc r1 v2)
| (n1,x1) :: r1, (n2,x2) :: r2 (* when n1 = n2 *)->
    cons (n1,(plus x1 x2)) (add plus np cons nc r1 r2)
| [], _ -> list_it (fun (n,x) y -> cons (n,(plus np x)) y) v2 nc
| _, [] -> list_it (fun (n,x) y -> cons (n,(plus x np)) y) v1 nc
;;

let (add_vect : vecteur -> vecteur -> vecteur) =
  add (prefix +.) 0. (fun (n,x) y -> if x <> 0. then (n,x) :: y else y) [] ;;
let (add_mat : matrice -> matrice -> matrice) =
  add add_vect [] (fun (n,x) y -> if x <> [] then (n,x) :: y else y) [] ;;

let scal = add (prefix *.) 0. (fun (n,x) y -> x +. y) 0. ;;

(*********************** Question 6 ***********************)

let element_v i v = scal [(i,1.)] v ;;

(*********************** Question 7 ***********************)
(* sans la fonction add mais avec une petite fonction auxillaire *)
```



```
let rec aux null i m = match m with
| (j,v) :: r when j > i -> null
| (j,v) :: r when j = i -> v
| (j,v) :: r -> aux null i r
| [] -> null
;;

let ligne i m = aux [] i m ;;

let element i j m =
  let v = ligne i m in
  aux 0. j v
;;

(* avec la fonction add *)
let ligne i m = add add_vect []
    (fun (n,x) y -> if n = i then x else y) []
    [] m;;

let element_m i j m = add scal2 []
    (fun (n,x) y -> if n = i then x else y) 0.
    [i,[j,1.]] m;;

(*********************** Question 8 ***********************)

let rec (add_col : int -> vecteur -> matrice -> matrice) =
  fun i v m -> match v,m with
  | [], _ -> m
  | (iv,x) :: rv, (im,vm) :: rm when iv=im ->
      (iv,(i,x) :: vm) :: (add_col i rv rm)
  | (iv,x) :: rv, (im,vm) :: rm when iv<im ->
      (iv,[i,x]) :: (add_col i rv m)
  | (iv,x) :: rv, (im,vm) :: rm (* when iv>im *) ->
      (im,vm) :: (add_col i v rm)
  | (iv,x) :: rv, [] ->
      (iv,[i,x]) :: (add_col i rv m)
;;

let (transpose : matrice -> matrice ) =
  fun m -> list_it (fun (n,v) -> add_col n v) m [] ;;

(*********************** Question 9 ***********************)

let rec (vprod : matrice -> vecteur -> vecteur) =
  fun m v -> match m with
  | [] -> []
  | (i, vm) :: r ->
      let tmp = scal vm v in
      if tmp <> 0. then
        (i, tmp) :: (vprod r v)
      else
        vprod r v
;;

(* avec la fonction de librairie ad hoc *)
let vprod m v =
  list_it (fun (i,vm) acc ->
    let t = scal vm v in
    if t = 0. then acc else (i,t) :: acc)
    m []
;;
```

```
(********************** Question 10 ***********************)

let mprod m1 m2 =
  let rec aux = function
    | [] -> []
    | (i, vm) :: r ->
        add_col i (vprod m1 vm) (aux r)
  in (aux (transpose m2) : matrice)
;;

(* avec la fonction de librairie ad hoc *)
let mprod m1 m2 =
  list_it (fun (i,vm) -> add_col i (vprod m1 vm)) (transpose m2) []
;;
```