```
(* ********************************************************************** *)
(* types sommes *)
type nombre = Entier of int | Flottant of float
let ajoute nombre1 nombre2 = match nombre1, nombre2 with
  | Entier a, Entier b -> Entier (a+b)
  | Flottant a, Flottant b -> Flottant (a+.b)
  | Entier _ , Flottant _ | Flottant _, Entier _ -> failwith "types incompatibles"

(* test *)
let _ = ajoute (Entier 4) (Entier 2)
let _ = ajoute (Entier 4) (Entier 3)
let _ = ajoute (Flottant 4.1) (Flottant 2.2)
let _ = ajoute (Flottant 4.) (Entier 2)

let divise nombre1 nombre2 = match nombre1 with
  | Entier n1 ->
      (match nombre2 with
         | Entier n2 ->
             if n1 mod n2 = 0
             then Entier (n1 / n2)
             else Flottant (float n1 /. float n2) (* [float] est un synonyme de [float_of_int]
 *)
         | Flottant f2 -> Flottant (float n1 /. f2))
  | Flottant f1 -> (* Il faut mettre le match precedent entre parentheses sinon ce cas lui app
artiendrait. *)
      match nombre2 with
        | Entier n2 -> Flottant (f1 /. float n2)
        | Flottant f2 -> Flottant (f1 /. f2)

(* test *)
let _ = divise (Entier 4) (Entier 2)
let _ = divise (Entier 4) (Entier 3)
let _ = divise (Flottant 4.) (Entier 2)

type 'a liste = Nil | Cons of 'a * 'a liste
let list_tl = function
  | Nil -> failwith "liste vide"
  | Cons (_,queue) -> queue

let list_rev liste =
  let entree = ref liste in
  let sortie = ref Nil in
  let continue = ref true in
  (* A chaque passage dans la boucle,
     on fait passer le premier element de [!entree] en tete de [!sortie]. *)
  while !continue do
    match !entree with
      | Nil -> continue := false
      | Cons (tete, queue) ->
          sortie := Cons (tete, !sortie);
          entree := queue;
  done;
  !sortie

(* autre version *)
let list_rev liste =
  let rec transvase entree sortie =
    (* Fait passer un a un les elements de [entree] dans [sortie].
       Equivalent a la boucle precedente. *)
    match entree with
      | Nil -> sortie
      | Cons (t,q) -> transvase q (Cons (t,sortie)) in
transvase liste Nil

let rec list_map f = function
  | Cons (tete, queue) -> Cons (f tete, list_map f queue)
  | Nil -> Nil

(* test *)
let liste_de_test = Cons (1, Cons (2,Cons(3,Nil)))
let _ = list_tl liste_de_test
let _ = list_rev liste_de_test
let _ = list_map float liste_de_test

(* ********************************************************************** *)
(* types enregistrement *)
type fraction = {numerateur:int; denominateur:int}
let ajoute fraction1 fraction2 =
  {denominateur = fraction1.denominateur * fraction2.denominateur;
   numerateur =
            fraction1.numerateur * fraction2.denominateur
            + fraction2.numerateur * fraction1.denominateur}

(* test *)
let _ = ajoute {numerateur = 1; denominateur = 2} {numerateur = 2; denominateur = 3}
```

```
(* version fonctionnelle *)
type 'a file = {debut:'a list; fin: 'a list}
let empty = {debut = []; fin = []}
let ajoute file element = {debut = file.debut; fin = element :: file.fin}
let rec prend file = match file.debut with
  | t :: q -> t, {debut = q; fin = file.fin}
  | [] -> match file.fin with
      | [] -> failwith "file vide" (* Cas ou les deux listes sont vides. *)
      | _ -> prend {debut = List.rev file.fin; fin = []}
          (* [prend] va choisir le premier cas du filtrage.
             Ce n'est pas vraiment une fonction recursive, c'est juste pour ne pas repeter du
code. *)
(* Fonctions symetriques: *)
let gruge file eleve = {file with debut = eleve :: file.debut}
let rec retire file = match file.fin with
  | t :: q -> t, {file with fin = q}
  | [] -> match file.debut with
      | [] -> failwith "file vide"
      | _ -> retire {debut = []; fin = List.rev file.debut}
(* test *)
let a = ajoute empty 2
let b = ajoute a 4
let c = ajoute b 5
let _ = retire c
let _ = prend c
let d = gruge c 1
let _ = prend d
let _ = retire d

(* version imperative *)
(* Cette fois il n'est plus necessaire de renvoyer la file car elle sera modifiee en place. *)
type 'a file = {mutable debut : 'a list; mutable fin : 'a list}
let create () = {debut = []; fin = []}
let ajoute file element = file.fin <- element :: file.fin
let rec prend file = match file.debut with
  | t :: q -> file.debut <- q; t
  | [] -> match file.fin with
      | [] -> failwith "file vide"
      | _ -> file.debut <- List.rev file.fin; file.fin <- []; prend file
let gruge file eleve = file.debut <- eleve :: file.debut
let rec retire file = match file.fin with
  | t :: q -> file.fin <- q; t
  | [] -> match file.debut with
      | [] -> failwith "file vide"
      | _ -> file.fin <- List.rev file.debut; file.debut <- []; retire file
(* test *)
let foo = create ();;
ajoute foo 0; foo;;
ajoute foo 2; foo;;
ajoute foo 4; foo;;
ajoute foo 5; foo;;
retire foo;;
foo;;
prend foo;;
foo;;
gruge foo 1; foo;;
prend foo;;
foo;;
retire foo;;
foo;;

(* question bonus *)
(* Renvoie la premiere moitie de la liste, et la seconde moitiee renversee. *)
let decoupe liste =
  let rec coupe_apres accu n lst = (* coupe [lst] apres [n] elements. *)
    if n = 0
    then List.rev accu, List.rev lst
    else match lst with
      | t::q -> coupe_apres (t::accu) (pred n) q
      | [] -> failwith "decoupe"
  in
  coupe_apres [] (List.length liste /2) liste

(* test *)
let _ = decoupe [1;2;3;4;5;6;7;8]

(* On modifie les deux fonctions voulues de la version fonctionnelle *)
let rec prend file = match file.debut with
  | t :: q -> t, {file with debut = q}
  | [] -> match file.fin with
      | [] -> failwith "file vide"
      | _ ->
          let fin, debut = decoupe file.fin in
          prend {debut = debut; fin = fin}
```

```
let rec retire file = match file.fin with
  | t :: q -> t, {file with fin = q}
  | [] -> match file.debut with
      | [] -> failwith "file vide"
      | _ ->
          let debut, fin = decoupe file.debut in
          retire {debut = debut; fin = fin}
```