```
(***************************************************************************)
(*                 Compression d'images par des arbres                     *)
(***************************************************************************)

type couleur = Blanc | Noir;;

type arbre = Feuille of couleur
           | Noeud   of arbre * arbre * arbre * arbre;;

type image == couleur vect vect;;

(* image -> arbre *)

let image_vers_arbre k t =
  let rec construit i j k =
    if k = 1 then
      Feuille t.(i).(j)
    else
      let k' = k / 2 in
      match
        construit i j k', construit (i+k') j k',
        construit i (j+k') k', construit (i+k') (j+k') k' with

          Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir ->
            Feuille Noir
        | Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc ->
            Feuille Blanc
        | c -> Noeud c
  in
    construit 0 0 k
;;

(* arbre -> image *)

let remplie_carre t i j k c =
  for x = i to i+k-1 do
    for y = j to j+k-1 do
      t.(x).(y) <- c
    done
  done
;;

let arbre_vers_image k a =
  let t = make_matrix k k Noir in
  let rec remplie i j k = function
      Feuille c ->
        remplie_carre t i j k c
    | Noeud (c1,c2,c3,c4) ->
        let k' = k / 2 in
          remplie i j k' c1 ;
          remplie (i+k') j k' c2 ;
          remplie i (j+k') k' c3 ;
          remplie (i+k') (j+k') k' c4
  in
  remplie 0 0 k a ; t
;;

(* arbre -> liste
   Remarque : on peut eviter les opérations de concaténation de listes (@)
              en construisant la liste à l'envers et en utilisant un
              accumulateur. *)

let arbre_vers_liste a =
```



```
  let rec construit = function
      Feuille Blanc ->
        [ `0` ; `0` ]
    | Feuille Noir ->
        [ `0` ; `1` ]
    | Noeud (a_1,a_2,a_3,a_4) ->
        `1` :: ((construit a_1) @ (construit a_2) @
                (construit a_3) @ (construit a_4))
  in
    construit a
;;

(* liste -> arbre *)

let liste_vers_arbre l =
  let rec fabrique = function
    | [ x ] ->
        failwith "liste mal formee"
    | `0` :: `0` :: l' -> Feuille Blanc, l'
    | `0` :: `1` :: l' -> Feuille Noir,  l'
    | `1` :: l0 ->
        let a_1,l1 = fabrique l0 in
        let a_2,l2 = fabrique l1 in
        let a_3,l3 = fabrique l2 in
        let a_4,l4 = fabrique l3 in
          Noeud (a_1,a_2,a_3,a_4),l4
    | _ -> failwith "liste contenant un caractere autre que 0 ou 1"
  in
    match fabrique l with
        a,[] -> a
      | _    -> failwith "liste trop longue"
;;

(* manipulation *)

let zoom a = a;; (* rien a faire ! *)

let rec rotation = function
    Noeud (a1,a2,a3,a4) ->
      Noeud (rotation a3, rotation a1, rotation a4, rotation a2)
  | x -> x
;;

(* affichage *)

#open "graphics";;

let dessine_arbre k a =
  let rec remplie i j k = function
      Feuille c ->
        set_color (match c with Blanc -> white | Noir -> black) ;
        fill_rect i j k k
    | Noeud (c1,c2,c3,c4) ->
        let k' = k / 2 in
          remplie i j k' c1 ;
          remplie (i+k') j k' c2 ;
          remplie i (j+k') k' c3 ;
          remplie (i+k') (j+k') k' c4
  in
  remplie 0 0 k a
;;

let rec fractale = function
```

```
    0 -> Feuille Noir
  | n ->
      let f = fractale (pred n) in
      let b = Feuille Blanc in
      Noeud (Noeud (f,f,f,b), Noeud (f,f,b,f),
             Noeud (f,b,f,f), Noeud (b,f,f,f))
;;

(* La fonction "fractale" est lineaire en temps et en espace.
   - en temps : car on a un appel récursif a (n−1) plus un temps constant ;
   - en espace: car on partage l’arbre obtenu par appel recursif (f)
                donc la taille pour n est la taille pour (n−1) plus une
                constante.
*)
```