Quadtrees et compression d'image

10/04/2020

Ce DM a été réalisé sur une configuration cygwin via l'éditeur top-ocaml sous Windows.

Développement:

MARTINS Romain

L2 INFO GRI2 TP B

Pour ce travail, nous considérons les types suivants :

```
type couleur =
    | Blanc
    | Noir

type quadtree =
    | Feuille of couleur
    | Noeud of quadtree * quadtree * quadtree * quadtree;;
```

Question 1

1. Écrire une fonction `quadtree_full` (respec. une fonction `quadtree_empty`) qui prend en argument un entier n tel que $n = 2^k$ et qui retourne un `quadtree` de hauteur k totalement noir (respec. totalement blanc).

On cherche donc à créer un quadtree de profondeur k+1 dont toutes les feuilles ont la même couleur : Noir pour quadtree_full et Blanc ou quadtree_empty.

Quadtree full :

```
let rec quadtree_full n =
  (*Return full black quadtree which the size is n x n
    val quadtree_full : int -> quadtree = <fun>*)

  if n mod 4 = 0 || n <= 2 then
    match n with
      (*Base case : smallest n = 2. 2x2 = Node(Leaf Black,Leaf Black,Leaf Black,Leaf Black)*)
      | 2 -> Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)
      (*Recursion : n = 2^k so we divide n by 2. Tree depth = k+1*)
      | a -> Noeud (quadtree_full (n/2),quadtree_full (n/2),quadtree_full (n/2),quadtree_full (n/2))

  (*Throw exception : Unsuitable size*)
  else
    failwith "Size error. Argument must be mutiple of 4 and arg>=2";;
```

On utilise la condition : n mod 4 || n <= 2 pour lever une exception dans le else au cas où la taille ne corresponde pas aux règles suivantes :

- Plus petit quadtree : n = 2
- Doit être divisible par n mod 4 (pour nos cas de test)

On sait que l'argument du modulo doit varier selon k (Pour traiter par exemple : n=12) mais cette condition n'a pas été obtenue.

Cas de base :

Notre cas de base prend n=2 soit un nœud dont les feuilles sont noires.

Récurrence :

```
(*Recursion : n = 2^k so we divide n by 2. Tree depth = k+1*)
| a -> Noeud (quadtree_full (n/2),quadtree_full (n/2),quadtree_full (n/2),quadtree_full (n/2))
```

Dans ce cas, nous ne sommes pas à la plus grande profondeur de l'arbre souhaitée, on crée donc un nœud dans lequel, pour chacun des 4 quadtrees, on rappelle la fonction ce qui nous retournera un arbre de même profondeur pour chaque branche.

Cas de tests : Initialisons un quadtree de taille 8, puis testons la levée d'exception.

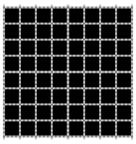
Ce test nous retourne :

```
# quadtree_full 8

- : quadtree =
Noeud
 (Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
 Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
 Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)),
 Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
  Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir)))

#
```

Soit une image de cette forme :



Retour du test d'exception :

```
# quadtree_full 15

Exception: Failure "Size error. Argument must be mutiple of 4 and arg>=2".

#
```

La fonction quadtree_empty fonctionne de la même façon mais les feuilles initialisées dans le cas de base sont de couleur blanche.

Quadtree_empty :

```
(*same code as for the previous question except that the quadtree is white*)

let rec quadtree_empty n =
  (*Return full white quadtree which the size is n x n
    val quadtree_full : int -> quadtree = <fun>*)
  if n mod 4 = 0 || n <= 2 then
    match n with
      | 2 -> Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc)
      | a -> Noeud (quadtree_empty (n/2),quadtree_empty (n/2),quadtree_empty (n/2),quadtree_empty (n/2))
  else
    failwith "Size error. Argument must be mutiple of 4 and arg>=2" ;;
```

Question 2

2. Écrire une fonction **inverse** qui prend un **quadtree a** représentant une image **i** et qui renvoie un **quadtree** représentant l'image **i'** obtenue à partir de **i** en échangeant **noir** et **blanc**.

Il s'agit ici d'échanger les couleurs de chaque point.

Inverse :

```
let rec inverse a =
  (*This function reverse leaf's colors.
    val inverse : quadtree -> quadtree = <fun>*)

  match a with
    (*Base cases : Return the reversed leaf's color*)
    | Feuille Noir -> Feuille Blanc
    | Feuille Blanc -> Feuille Noir

    (*Recursion : Course each node for each leaf to apply the base case.*)
    | Noeud (w,x,y,z) -> Noeud (inverse w, inverse x, inverse y, inverse z);;
```

Cas de base :

Le type Couleur ne comprend pas de couleur 'par défaut'. Il s'agit donc simplement d'échanger noir et blanc.

Récurrence :

On s'assure que la fonction soit appliquée à toutes les feuilles en l'appelant à chaque quadtree du nœud courant.

Cas de test : On crée d'abord un quadtree non monochrome de test.

```
(*First quadtree_test's initialization*)
let quadtree_test =
  Noeud (
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Noir),
    Noeud (Feuille Blanc,Feuille Noir,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Blanc,Feuille Blanc,Feuille Noir,Feuille Blanc)
  );;

(*Test*)
inverse quadtree_test;;
```

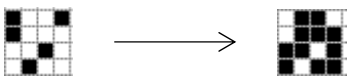
Cette fonction retourne donc :

```
# inverse quadtree_test

- : quadtree =
Noeud (Noeud (Feuille Blanc, Feuille Noir, Feuille Noir, Feuille Blanc),
 Noeud (Feuille Noir, Feuille Blanc, Feuille Noir, Feuille Noir),
 Noeud (Feuille Blanc, Feuille Noir, Feuille Noir, Feuille Noir),
 Noeud (Feuille Noir, Feuille Noir, Feuille Blanc, Feuille Noir))

#
```

Illustration :



Question 3

3. Écrire une fonction **rotate** qui prend un **quadtree** a représentant une image i et qui renvoie un **quadtree** représentant l'image i tournée d'un quart de tour vers la gauche.

Rotate :

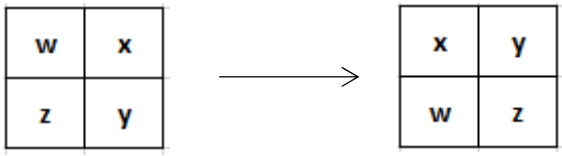
```
let rec rotate quadtree =
  (*This function turn left a quadtree.
    val rotate : quadtree -> quadtree = <fun>*)

  match quadtree with
    (*Base cases : Each Leaf go back one index.*)
    | Feuille a -> Feuille a
    | Noeud (Feuille w,Feuille x,Feuille y,Feuille z) -> Noeud(Feuille x,Feuille y,Feuille z,Feuille w)

    (*Recursion : Each node go back one index in quadtree parameter (0,1,2,3) -> (1,2,3,0)*)
    | Noeud (w,x,y,z) -> Noeud (rotate x,rotate y,rotate z,rotate w);;
```

Cas de base :

Ici, la rotation d'un quart vers la gauche se traduit par reculer les feuilles d'un index selon le schéma suivant.



Récurrence :

On tourne également d'un quart tous les nœuds respectant là aussi le schéma ci-dessus.

Cas de test :

On utilise le quadtree_test suivant :

```
let quadtree_test =
  Noeud (
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Noir),
    Noeud (Feuille Blanc,Feuille Noir,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Blanc,Feuille Blanc,Feuille Noir,Feuille Blanc)
  );;
```

Sortie :

```
# rotate quadtree_test

- : quadtree =
Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Blanc, Feuille Blanc),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Noir),
 Noeud (Feuille Blanc, Feuille Noir, Feuille Blanc, Feuille Blanc),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Noir, Feuille Noir))

#
```

Illustration :

Question 4

4. Écrire une fonction `union` qui prend deux `quadtree` `a` et `b` représentant chacun respectivement l'image `i` et l'image `i'` et qui renvoie un `quadtree` `c` représentant l'union des deux images.

Pour union, nous respectons les conventions logiques suivantes :

union (∪)
blanc ∪ x = x ∪ blanc = x
noir ∪ noir = noir

Union :

```
let rec union a b =
  (*This function create new quadtree in accordance with a and b as these rules : Blanc U x = x U Blanc = x ; Noir U Noir = Noir
    val union : quadtree -> quadtree -> quadtree = <fun>*)

  match a,b with
    (*Throw exception when quadtrees a and b have differents sizes.*)
    | Noeud (_,_,_,_) , Feuille _
    | Feuille _ , Noeud (_,_,_,_) -> failwith "Quadtrees have differents sizes."

    (*Base cases : Apply rules*)
    | a,Feuille Blanc | Feuille Blanc, a -> a
    | Feuille Noir,Feuille Noir -> Feuille Noir

    (*Recursion : Course a and b at same level and apply union*)
    | Noeud (a0,a1,a2,a3),Noeud (b0,b1,b2,b3) -> Noeud (union a0 b0, union a1 b1, union a2 b2, union a3 b3);;
```

Exception :

On lève une exception lorsque les quadtrees sont de tailles différentes, si les profondeurs sont différentes, un message s'affiche et la fonction s'arrête.

Cas de base :

On traite en respectant les conventions logiques précédemment citées.

Récurrence :

On réalise l'union des quadtrees a et b en choisissant les arguments pour rester à la même profondeur dans les mêmes quadtrees sur a et sur b sans changer leurs localisations.

Cas de test :

Test d'exception :

a

```
let quadtree_unionException = quadtree_empty 8;; (*quadtree_test's size is 16 (16x16)*)
```

b

```
let quadtree_test =
  Noeud (
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Noir),
    Noeud (Feuille Blanc,Feuille Noir,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Blanc,Feuille Blanc,Feuille Noir,Feuille Blanc)
  );;
```

Sortie

```
# union quadtree_test quadtree_unionException
Exception: Failure "Quadtrees have differents sizes.".
#
```

On utilise ensuite quadtree_test (précédemment b) et le quadtree suivant pour exécuter la fonction.

```
# let i' = rotate quadtree_test

val i' : quadtree =
  Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Blanc, Feuille Blanc),
   Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Noir),
   Noeud (Feuille Blanc, Feuille Noir, Feuille Blanc, Feuille Blanc),
   Noeud (Feuille Blanc, Feuille Blanc, Feuille Noir, Feuille Noir))

#
```

Sortie :

```
# union quadtree_test i'

- : quadtree =
Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Blanc, Feuille Noir),
 Noeud (Feuille Blanc, Feuille Noir, Feuille Blanc, Feuille Noir),
 Noeud (Feuille Noir, Feuille Noir, Feuille Blanc, Feuille Blanc),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Noir, Feuille Noir))

#
```

Illustration :

Question 5

5. Écrire une fonction **intersection** qui prend deux **quadtree** **a** et **b** représentant chacun respectivement l'image **i** et l'image **i'** et qui renvoie un **quadtree** **c** représentant l'intersection des deux images.

Cette fonction requière également des conventions logiques :

intersection (∩)
blanc ∩ x = x ∩ blanc = blanc
noir ∩ noir = noir

Intersection :

```
let rec intersection a b =
  (*This function create new quadtree in accordance with a and b as these rules : Blanc U x = x U Blanc = Blanc ; Noir U Noir = Noir
    val intersection : quadtree -> quadtree -> quadtree = <fun>*)

  match a,b with
    (*Throw exception when quadtrees a and b have differents sizes.*)
    | Noeud (_,_,_,_) , Feuille _
    | Feuille _ , Noeud (_,_,_,_) -> failwith "Quadtrees have differents sizes."

    (*Base cases : Apply rules*)
    | a,Feuille Blanc | Feuille Blanc, a -> Feuille Blanc
    | Feuille Noir,Feuille Noir -> Feuille Noir

    (*Recursion : Course a and b at same level and apply intersection*)
    | Noeud (a0,a1,a2,a3),Noeud (b0,b1,b2,b3) -> Noeud (intersection a0 b0, intersection a1 b1, intersection a2 b2, intersection a3 b3);;
```

Général :

Seul le cas de base change selon les conventions logiques ci-dessus (qui sont donc appliquées dans les cas de base), le reste de la fonction est identique à la question 1.

Cas de test :

On utilise les mêmes a et b que précédemment

a

```
let quadtree_test =
  Noeud (
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Noir),
    Noeud (Feuille Blanc,Feuille Noir,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Blanc,Feuille Blanc,Feuille Noir,Feuille Blanc)
  );;
```

b

```
# let i' = rotate quadtree_test

val i' : quadtree =
  Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Blanc, Feuille Blanc),
    Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Noir),
    Noeud (Feuille Blanc, Feuille Noir, Feuille Blanc, Feuille Blanc),
    Noeud (Feuille Blanc, Feuille Blanc, Feuille Noir, Feuille Noir))
#
```

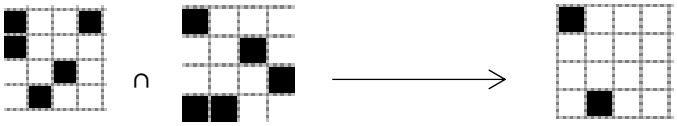
Sortie :

```
# intersection quadtree_test i'

- : quadtree =
Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Blanc, Feuille Blanc),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Noir, Feuille Blanc))

#
```

Illustration :

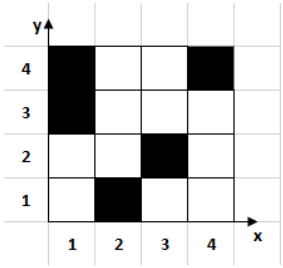


Question 6

6. Écrire une fonction `color` qui prend en arguments les coordonnées `(x, y)` d'un point dans l'image `i`, la taille de `i` et le quadtree qui lui correspond et qui retourne la couleur du point. Si les coordonnées du point sont incorrectes, la fonction retourne la valeur `None`.

On considère (x,y) comme des coordonnées d'un repère orthonormé sur lequel serait l'image.

Ainsi :



Color :

```
let rec color (x,y) taille quadtree =
  (*This function return the leaf's color in (x,y) coordinates.
    val color : int * int -> int -> quadtree -> quadtree = <fun>*)

  (*Throw an exception when x or y exceed the picture size*)
  if x > taille || x <= 0 || y > taille || y <= 0 then
    failwith "Please enter coordinates between 1 and the quadtree's size."
  else
    match quadtree with

      (*Base cases : Return quatree*)
      | Feuille _ -> quadtree

      (*Recursion : identifies the quadtree's quarter concerned (a,b,c,d) and call color with this new node.
        With each new call, the quadtree's size is devided by 2, if x>(taille/2), we call color with x-(taille/2). Same for y.*)
      | Noeud (a,b,c,d) ->
          if x <= (taille/2) && y <= (taille/2) then
            color (x,y) (taille/2) d
          else if x <= (taille/2) && y > (taille/2) then
            color (x,(y-(taille/2))) (taille/2) a
          else if x > (taille/2) && y <= (taille/2) then
            color ((x-(taille/2)),y) (taille/2) c
          else
            color ((x-(taille/2)),(y-(taille/2))) (taille/2) b;;
```

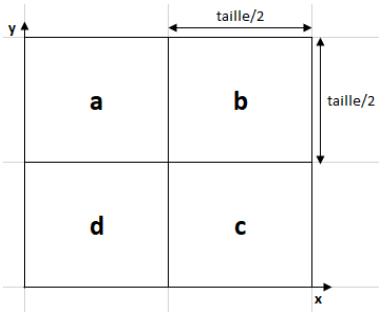
On lève une exception si les coordonnées demandées dépassent celles de l'image, la valeur de retour est alors définie à None

Cas de base :

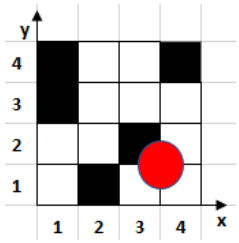
L'arbre a été parcouru, on retourne la couleur de la feuille.

Récurrence :

On cherche à parcourir l'arbre en suivant cette méthode :



Exemple :



Si on prend (3,1) avec un quadtree de taille 4. Alors,

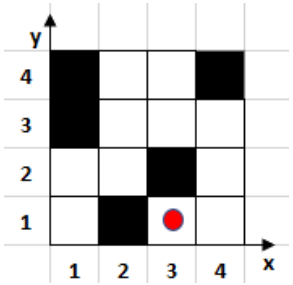
x >= (taille/2) et y <= (taille/2), on sait que (3,1) se situe dans le quadtree c.

On appelle la fonction avec ce nouveau quadtree et une taille de moitié plus petite, ainsi le coin bas/gauche de ce quadtree prendra les coordonnées (1,1). Ainsi, si le repère a changé pour y alors sa nouvelle valeur est y – (taille/2).

Dans notre exemple, on a maintenant (1,1), la taille divisée par 2 et le quadtree précédemment c.

X <= (taille/2) et y <= (taille/2) donc nos coordonnées se situent dans le quadtree d.

Ce quadtree d est une feuille, le prochain appel passe par le cas de base et renvoie donc Blanc par récursivité terminale.



Cas de test :

Test d'exception :

```
# color (-2,5) 4 quadtree_test

Exception:
Failure "Please enter coordinates between 1 and the quadtree's size.".

#
```

Testons la fonction avec un quadtree dont seule une feuille est blanche et exécutons.

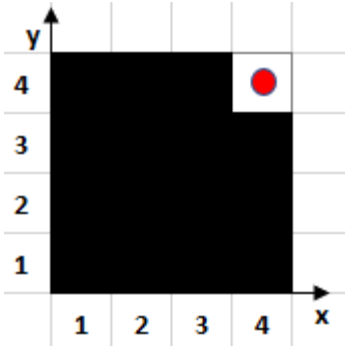
```
val quadtree_test_color : quadtree =
  Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Blanc, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir))

# color (4,4) 4 quadtree_test_color

- : couleur = Blanc

#
```

Illustration :



Question 7

7. Écrire une fonction `modify` qui prend un `quadtree a` représentant une image `i` et la taille de `i` et qui modifie `i` en appliquant à chacun de ses points une fonction de profil `int -> int -> couleur -> couleur`. L'argument de type `couleur` est l'ancienne couleur du point, et la fonction retourne sa nouvelle couleur.

Modify :

```
let rec modify quadtree current_x current_y fonction taille =
  (*Modify Leaf color after testing coordinates (in fonction)
    int -> int -> (int -> int -> couleur -> couleur) -> int -> quadtree = <fun>*)
  match quadtree with
    (*Base case : We apply fonction with current x,y,color to test it in the fonction in argument*)
    | Feuille e -> Feuille (fonction current_x current_y e)
    | Noeud (a,b,c,d)
     (*Course the tree by increasing x,y as it was the picture reconsitution in an orthonormal coordinate system*)
     ->  Noeud (modify a current_x (current_y+taille/2) fonction (taille/2),
               modify b (current_x+taille/2) (current_y+taille/2) fonction (taille/2),
               modify c (current_x+taille/2) current_y fonction (taille/2),
               modify d current_x current_y fonction (taille/2));;
```

Nous ajoutons ici les variables current_x et current_y qui serviront lors des appels récursifs à connaître les coordonnées courantes, toujours selon un repère orthonormé.

Cas de base :

On applique la fonction en paramètre à la feuille quadtree.

Récurrence :

Nous gérons les coordonnées d'une manière similaire à la question 6, sauf qu'on les incrémente lors de l'appel plutôt que de les décrémenter. On appelle modify à chaque quadtree du nœud. Ainsi :

- Pour a : x = x et y = y + (taille/2)
- Pour b : x = x + (taille/2) et y = y + (taille/2)
- Pour c : x = x + (taille/2) et y = y
- Pour d : x = x et y = y

La fonction en argument comprendra donc un test sur x,y,couleur e. Si ces trois conditions sont validées, on change la couleur.

```
(fun x y couleur_ini -> if x=3 && y=1 && couleur_ini = Noir then Blanc else couleur_ini)
```

Dans cet exemple x=3, y=1 couleur_ini=Noir et la couleur que l'on souhaite est Blanc.

Cas de test : Prenons un quadtree entièrement noir.

```
# modify quadtree_test_modify 1 1 (fun x y couleur_ini -> if x=3 && y=1

- : quadtree =
Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
 Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
 Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Blanc),
 Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir))

#
```

Illustration :

Question 8

8. Puisqu'on peut modifier une image, écrire une fonction `optimise` qui prend le `quadtree` a qui lui correspond et qui optimise sa représentation arborescente : un nœud dont les fils sont tous de la même couleur sera modifié en feuille.

Optimise :

```
let rec optimise quadtree =
  (*This function return an optimal quadtree (Noeud(Feuille Noir,Feuille Noir,Feuille Noir,Feuille Noir) = Feuille Noir)
    val optimise : quadtree -> quadtree = <fun>*)
  match quadtree with
    (*Base cases : Return leaf
      Leaves are joined as in a function's description*)
    | Feuille e -> Feuille e
    | Noeud (Feuille Noir,Feuille Noir,Feuille Noir,Feuille Noir) -> Feuille Noir
    | Noeud (Feuille Blanc,Feuille Blanc,Feuille Blanc,Feuille Blanc) -> Feuille Blanc
    (*Recursion : Visit all of leaves*)
    | Noeud (a,b,c,d) ->
        if Noeud (optimise a, optimise b, optimise c, optimise d) = Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir) then
          Feuille Noir
        else if Noeud (optimise a, optimise b, optimise c, optimise d) = Noeud (Feuille Blanc, Feuille Blanc, Feuille Blanc, Feuille Blanc) then
          Feuille Blanc
        else
          (* We test it after returning from function to verify if new node is monochrome*)
          Noeud (optimise a, optimise b, optimise c, optimise d);;
```

Cas de base :

- On est à profondeur k+1, on renvoie la feuille
- Pattern où les feuilles du nœud sont de couleur identique, on renvoie une feuille de cette couleur

Récurrence :

Dans le cas où nous pourrions optimiser deux profondeurs successives de l'arbre, nous appelons la fonction optimise, afin de tester ensuite si les nouvelles feuilles sont de couleur identique.

Si oui, on les remplace par une unique feuille de couleur.

Cas de test : Prenons un quadtree polychrome et un autre monochrome.

```
val quadtree_test_optimise : quadtree =
  Noeud (Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Blanc, Feuille Noir, Feuille Noir, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Blanc, Feuille Noir),
   Noeud (Feuille Noir, Feuille Noir, Feuille Noir, Feuille Noir))
```

Sortie :

```
# optimise quadtree_test_optimise

- : quadtree =
Noeud (Feuille Noir,
 Noeud (Feuille Blanc, Feuille Noir, Feuille Noir, Feuille Noir),
 Noeud (Feuille Noir, Feuille Noir, Feuille Blanc, Feuille Noir),
 Feuille Noir)

#
```

Cas monochrome :

```
let quadtree_test_empty_optimise = quadtree_empty 8;;
```

Sortie :

```
# optimise quadtree_test_empty_optimise

- : quadtree = Feuille Blanc

#
```

Question 9

9. Écrire une fonction `quadtree_to_list` de type `quadtree -> bit list` qui transforme un `quadtree` en une liste de bits selon le codage.

Considérons le type suivant : `type bit = Zero | Un`

Nous suivrons les conventions logiques suivantes :

```
code(Feuille Blanc) = 00
code(Feuille Noir) = 01
code(Noeud (a1,a2,a3,a4)) = 1 code(a1) code(a2) code(a3) code(a4)
```

Quadtree_to_list :

```
let rec quadtree_to_list quadtree =
  (*Converts quadtree to bit list
    val quadtree_to_list : quadtree -> bit list = <fun>*)
  match quadtree with
    (*base cases : We use bit type and homogenize them*)
    | Feuille Noir -> Zero::Un::[]
    | Feuille Blanc -> Zero::Zero::[]
    (**)
    | Noeud (a,b,c,d) -> Un::[] @ quadtree_to_list a @ quadtree_to_list b @ quadtree_to_list c @ quadtree_to_list d;;
```

Cas de base :

Nous appliquons les conventions ci-dessus sous forme de liste pour pouvoir les concaténer lors de la récursion et pour respecter les règles de typage d'Ocaml.

Récurrence :

On ajoute Un en tête selon les conventions puis concaténons les appels sur les 4 quadtrees du nœud.

Cas de test :

```
let quadtree_test =
  Noeud (
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Noir),
    Noeud (Feuille Blanc,Feuille Noir,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Blanc),
    Noeud (Feuille Blanc,Feuille Blanc,Feuille Noir,Feuille Blanc));;
```

Sortie :

```
- : bit list =
[Un; Un; Zero; Un; Zero; Zero; Zero; Zero; Zero; Un; Un; Zero; Zero; Zero;
 Un; Zero; Zero; Zero; Zero; Un; Zero; Un; Zero; Zero; Zero; Zero; Zero;
 Zero; Un; Zero; Zero; Zero; Zero; Zero; Un; Zero; Zero]

#
```

Question 10

10. Écrire une fonction de décodage `list_to_quadtree` de type `bit list -> quadtree` qui transforme une liste de bits en le `quadtree` correspondant. Le quadtree devra être optimal. Ne pas utiliser la fonction `optimise` pour écrire cette fonction.

Ici, nous tentons de faire le chemin inverse. Cependant cette fonction ne retourne pas le résultat attendu.

List_to_quadtree :

```
let rec list_to_quadtree liste =
  match liste with
      [] -> failwith"Error"
    | Zero::Zero::t -> Feuille Blanc
    | Zero::Un::t -> Feuille Noir
    | Un::first -> match first with
      | h::i::second ->
          match second with
            | h::i::third ->
              match third with
                | h::i::fourth ->
                    Noeud(list_to_quadtree first, list_to_quadtree second, list_to_quadtree third, list_to_quadtree fourth);;
```

Cas de base :

A l'inverse de la fonction précédente, ici [0 ;0] return une feuille blanche et [0 ;1] une feuille noire.

Récurrence :

Si on ne retrouve pas ces patterns, alors on sait qu'il s'agit d'un nœud, pour pouvoir utiliser le constructeur Nœud, nous devons avoir les quadtrees qui le composent (raison pour laquelle cette solution est proposée malgré son manque d'élégance).

N'étant pas parvenu à identifier comment parcourir la liste de sorte à utiliser les index représentants les feuilles voulues, cette proposition compile néanmoins (et malgré quelques patterns non traités).

Cas de test :

```
- : bit list =
[Un; Un; Zero; Un; Zero; Zero; Zero; Zero; Zero; Un; Un; Zero; Zero; Zero;
 Un; Zero; Zero; Zero; Zero; Un; Zero; Un; Zero; Zero; Zero; Zero; Zero;
 Zero; Un; Zero; Zero; Zero; Zero; Zero; Un; Zero; Zero]

#
```

Résultat obtenu :

```
# list_to_quadtree list_test

- : quadtree =
Noeud (Noeud (Feuille Noir, Feuille Blanc, Feuille Blanc, Feuille Noir),
 Noeud (Feuille Blanc, Feuille Blanc, Feuille Noir,
  Noeud (Feuille Blanc, Feuille Noir, Feuille Blanc, Feuille Blanc)),
 Feuille Blanc, Feuille Blanc)

#
```

Ces feuilles sont donc une répétition de la première suite de feuille représentée par la liste.

Résultat attendu :

```
# let quadtree_test =
    Noeud (
      Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Noir),
      Noeud (Feuille Blanc,Feuille Noir,Feuille Blanc,Feuille Blanc),
      Noeud (Feuille Noir,Feuille Blanc,Feuille Blanc,Feuille Blanc),
      Noeud (Feuille Blanc,Feuille Blanc,Feuille Noir,Feuille Blanc))
```

Conseils d'utilisation

Aucune interface d'affichage n'a été implémentée, il s'agit donc de tester manuellement via emacs.

Quadtree_full & quadtree_empty :

Entrer une taille n=2^k.

Inverse & rotate & optimise & quadtree_to_list :

Entrer simplement un quadtree valide.

Union & Intersection :

Entrer deux quadtrees valides de taille identique.

Color :

Entrer respectivement un 2-uplet de coordonnées (x,y) avec 1 <= x <= taille et 1 <= y <= taille[1], la taille puis un quadtree valide.

Modify :

Entrer respectivement un quadtree valide, les valeur 1 et 1, une fonction vérifiant x,y[2] et la couleur souhaitée puis la taille du quadtree.

```
modify quadtree_test_modify 1 1 (fun x y couleur_ini -> if x=3 && y=1 && couleur_ini = Noir then Blanc else couleur_ini) 4;;
```

List_to_quadtree :

Entrer une liste respectant les conventions logiques suivantes :

```
code(Feuille Blanc) = 00
code(Feuille Noir) = 01
code(Noeud (a1,a2,a3,a4)) = 1 code(a1) code(a2) code(a3) code(a4)
```

[1] : Le choix de bordure pour les coordonnées est ainsi pour rendre plus intuitif l'utilisation à un utilisateur non averti dans le cas d'une création d'interface (qui n'a finalement pas été réalisée).

[2] : Les coordonnées respectent toujours les intervalles suivants : 1<=x<=taille & 1<=y<=taille

Répertoire des entrées/sorties

val quadtree_full : int -> quadtree

val quadtree_empty : int -> quadtree

val inverse : quadtree -> quadtree

val quadtree_test : quadtree

val rotate : quadtree -> quadtree

val union : quadtree -> quadtree -> quadtree

val i' : quadtree

val quadtree_unionException : quadtree

val intersection : quadtree -> quadtree -> quadtree

val color : int * int -> int -> quadtree -> couleur

val quadtree_test_color : quadtree

val modify :

quadtree ->

int -> int -> (int -> int -> couleur -> couleur) -> int -> quadtree

val quadtree_test_modify : quadtree

val optimise : quadtree -> quadtree

val quadtree_test_optimise : quadtree

val quadtree_test_empty_optimise : quadtree

val quadtree_to_list : quadtree -> bit list

val list_to_quadtree : bit list -> quadtree

val list_test : bit list

Ces détails sont présents dans le fichier MartinsRomainDM.mli, dans le premier commentaire de chaque fonction dans le rapport et dans le code source.