Programmation fonctionnelle pour le web

II – Types algébriques

Guillaume Geoffroy
guillaume.geoffroy@irif.fr

UFR d'Informatique

L1 2023–2024

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
type type_b = D of type_a | E | F of type_a * bool
```

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```
10

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```
10

Exercice : avec paramètres

En supposant que `type_x` est un type avec `x` valeurs différentes, déterminer le nombre de valeurs différentes qui existent pour les types `type_x t`, avec `'a t` les types suivants.

1. `type 'a option = None | Some of 'a`
2. `type 'a type_c = G of ('a option) * ('a option)`
3. `type 'a type_d = H of 'a * 'a | I of 'a * bool | J`

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```
10

Exercice : avec paramètres

En supposant que `type_x` est un type avec `x` valeurs différentes, déterminer le nombre de valeurs différentes qui existent pour les types `type_x t`, avec `'a t` les types suivants.

1. `type 'a option = None | Some of 'a` — $x + 1$
2. `type 'a type_c = G of ('a option) * ('a option)`
3. `type 'a type_d = H of 'a * 'a | I of 'a * bool | J`

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```
10

Exercice : avec paramètres

En supposant que `type_x` est un type avec `x` valeurs différentes, déterminer le nombre de valeurs différentes qui existent pour les types `type_x t`, avec `'a t` les types suivants.

1. `type 'a option = None | Some of 'a` — $x + 1$
2. `type 'a type_c = G of ('a option) * ('a option)` — $(x + 1)^2$
3. `type 'a type_d = H of 'a * 'a | I of 'a * bool | J`

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```
10

Exercice : avec paramètres

En supposant que `type_x` est un type avec `x` valeurs différentes, déterminer le nombre de valeurs différentes qui existent pour les types `type_x t`, avec `'a t` les types suivants.

1. `type 'a option = None | Some of 'a` — $x + 1$
2. `type 'a type_c = G of ('a option) * ('a option)` — $(x + 1)^2$
3. `type 'a type_d = H of 'a * 'a | I of 'a * bool | J` — $x^2 + 2x + 1$

Retour sur les types sommes

Exercice : sans paramèteres

Déterminer combien de valeurs différentes existent pour chacun des types suivants.

```
type type_a = A | B | C
```
3

```
type type_b = D of type_a | E | F of type_a * bool
```
10

Exercice : avec paramètres

En supposant que `type_x` est un type avec `x` valeurs différentes, déterminer le nombre de valeurs différentes qui existent pour les types `type_x t`, avec `'a t` les types suivants.

1. `type 'a option = None | Some of 'a` — $x+1$
2. `type 'a type_c = G of ('a option) * ('a option)` — $(x+1)^2$
3. `type 'a type_d = H of 'a * 'a | I of 'a * bool | J` — x^2+2x+1

Exercice : bijections

Écrire deux fonctions
```
d_from_c : 'a type_c -> 'a type_d
c_from_d : 'a type_d -> 'a type_c
```
qui définissent pour tout `'a` des bijections réciproques entre `'a type_c` et `'a type_d`.

Retour sur les types sommes

Exercice : bijections

Écrire deux fonctions `d_from_c : 'a type_c -> 'a type_d`
`c_from_d : 'a type_d -> 'a type_c`
qui définissent pour tout `'a` des bijections réciproques entre `'a type_c` et `'a type_d`.

```
let d_from_c = function
  | G (Some x, Some y) -> H (x, y)
  | G (Some x, None) -> I (x, false)
  | G (None, Some y) -> I (y, true)
  | G (None, None) -> J

let c_from_d = function
  | H (x, y) -> G (Some x, Some y)
  | I (x, false) -> G (Some x, None)
  | I (y, true) -> G (None, Some y)
  | J -> G (None, None)
```

Retour sur les listes

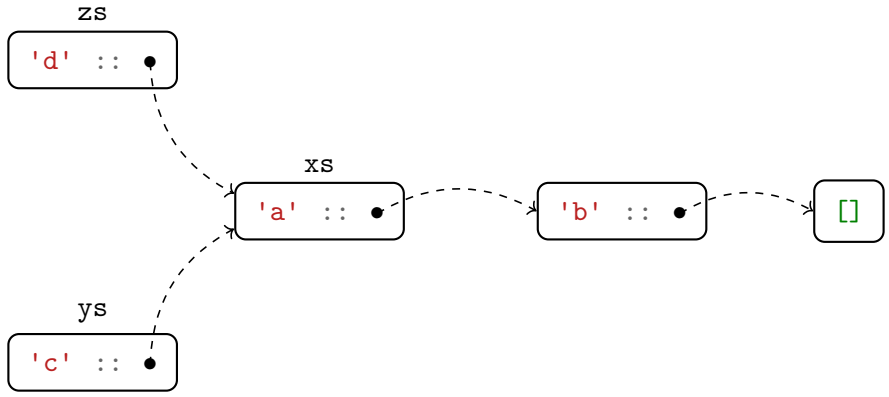
```
type liste_entiers =
  | Aucun
  | UnDePlus of int * liste_entiers

let xs =                          (* : liste_entiers *)
  UnDePlus (1, UnDePlus (2, UnDePlus (3, Aucun)))

(*********************************************************)

type 'a list =
  | []
  | ( :: ) of 'a * 'a list

let xs = 1 :: (2 :: (3 :: []))   (* : int list *)

let xs = [1; 2; 3]               (* : int list *)
```

Une représentation visuelle des listes en mémoire

```
let xs = 'a' :: 'b' :: [] (* ['a' ; 'b'] *)
let ys = 'c' :: xs
let zs = 'd' :: xs
```

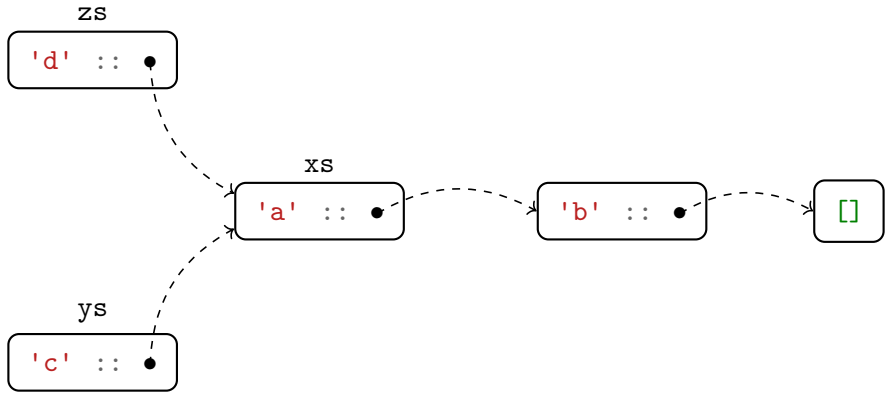
Une représentation visuelle des listes en mémoire

```
let xs = 'a' :: 'b' :: [] (* ['a' ; 'b'] *)
let ys = 'c' :: xs
let zs = 'd' :: xs
```



Une représentation visuelle des listes en mémoire

```
let xs = 'a' :: 'b' :: [] (* ['a' ; 'b'] *)
let ys = 'c' :: xs
let zs = 'd' :: xs
```



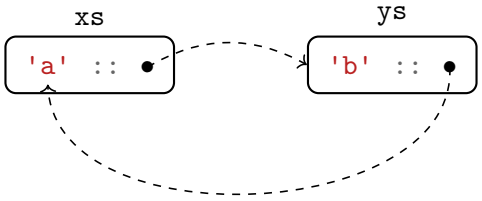
Une valeur de type `'a list` est donc bien une *liste* (suite) de nœuds.

Une représentation visuelle des listes en mémoire

```
let rec xs = 'a' :: ys
and      ys = 'b' :: xs
```

Une représentation visuelle des listes en mémoire

```
let rec xs = 'a' :: ys
and      ys = 'b' :: xs
```



Les types algébriques, en général

```
type ('a1, ..., 'aL) nom = K_1 of t_1_1 * ... * t_1_M1
                         | ...
                         | K_N of t_N_1 * ... * t_N_MN
```

Les définitions

- Explicites : énumèrent les *constructeurs* des valeurs du type `nom`.
- Chaque constructeur spécifie un nombre fini de *paramètres*.
- Le type d'un paramètre peut être celui en train d'être défini !
- Les programmes distinguent les constructeurs via le filtrage (`match`).

Terminologie

On parle de type "algébrique" parce qu'on a une **somme** de **produits**.

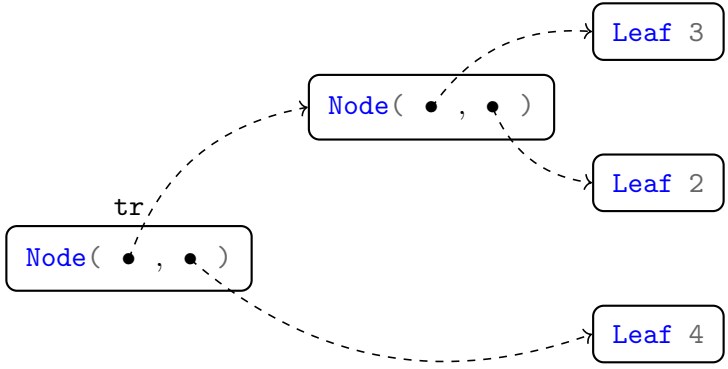
- La situation évoque celle des polynômes en mathématiques.
- On parle aussi de *types sommes*, ou plus rarement de *types variants*.

Une représentation visuelle d'un type d'arbres en mémoire

```
type bintree = Leaf of int | Node of bintree * bintree

let tr = Node (Node (Leaf 3, Leaf 2), Leaf 4)
```

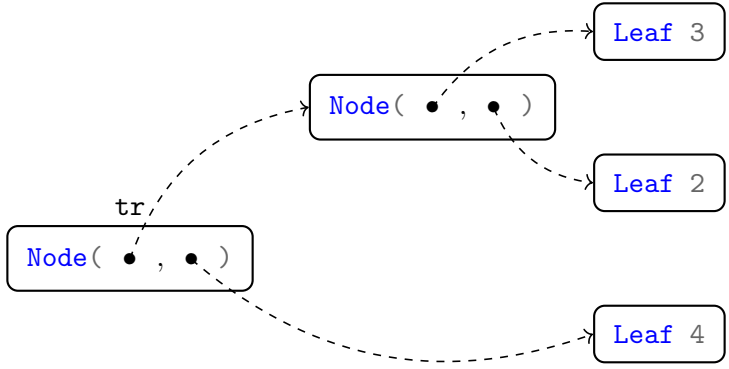
Une représentation visuelle d'un type d'arbres en mémoire

```
type bintree = Leaf of int | Node of bintree * bintree

let tr = Node (Node (Leaf 3, Leaf 2), Leaf 4)
```



Une représentation visuelle d'un type d'arbres en mémoire

```
type bintree = Leaf of int | Node of bintree * bintree

let tr = Node (Node (Leaf 3, Leaf 2), Leaf 4)
```



Une valeur de type `bintree` est donc bien un *arbre* d'entiers.

Une représentation visuelle d'un type d'arbres en mémoire

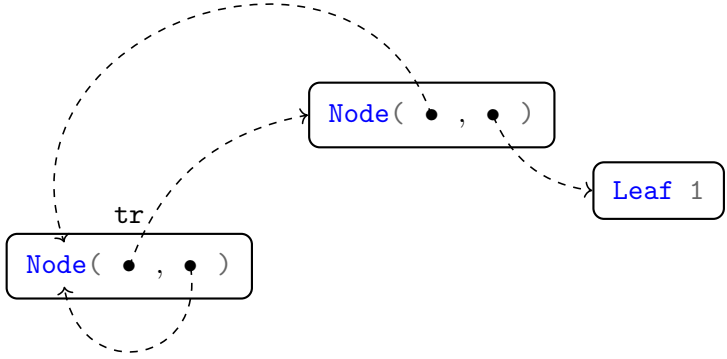
```
let rec tr = Node (Node (tr, Leaf 1), tr)
```

Exercice

Dessiner la représentation en mémoire de `tr`

Une représentation visuelle d'un type d'arbres en mémoire

```
let rec tr = Node (Node (tr, Leaf 1), tr)
```



Objectifs de cette séance

- Se familiariser avec les types algébriques au-delà des listes.
- Vous connaissez déjà les constructions requises (filtrage, récursion).
- Il s'agira donc essentiellement de pratiquer !

Arbres quaternaires

Exercice

Définir un type `'a quadtree` permettant de représenter des arbres quaternaires (chaque nœud a 4 fils) dont les feuilles portent des étiquettes de type `'a`.

Arbres quaternaires

Exercice

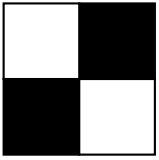
Définir un type `'a quadtree` permettant de représenter des arbres quaternaires (chaque nœud a 4 fils) dont les feuilles portent des étiquettes de type `'a`.

```
type 'a quadtree =
   Leaf of 'a
 | Node of 'a quadtree * 'a quadtree * 'a quadtree * 'a quadtree
```

Images par arbres quaternaires

On utilise le type suivant pour représenter des images de carrées en noir et blanc :

```
type color = Black | White
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```



```
Square {nw = Uniform White;
        ne = Uniform Black;
        se = Uniform White;
        sw = Uniform Black}
```

Images par arbres quaternaires - Exercice 1

```
type color = Black | White
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Dessiner les images représentés par les valeurs des variables ci-dessous.

1. `let q1 = Square { nw = black; ne = white; se = white; sw = black }`
2. `let q2 = Square { nw = black; ne = black; se = black; sw = white }`
3. `let q3 = Square { nw = white; ne = black; se = white; sw = q2 }`

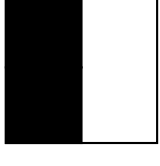
Images par arbres quaternaires - Exercice 1

```
type color = Black | White
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Dessiner les images représentés par les valeurs des variables ci-dessous.

1. `let q1 = Square { nw = black; ne = white; se = white; sw = black }`
2. `let q2 = Square { nw = black; ne = black; se = black; sw = white }`
3. `let q3 = Square { nw = white; ne = black; se = white; sw = q2 }`



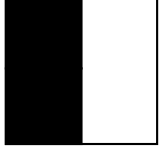
Images par arbres quaternaires - Exercice 1

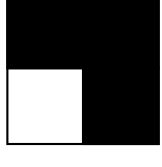
```
type color = Black | White
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Dessiner les images représentés par les valeurs des variables ci-dessous.

1. `let q1 = Square { nw = black; ne = white; se = white; sw = black }`
2. `let q2 = Square { nw = black; ne = black; se = black; sw = white }`
3. `let q3 = Square { nw = white; ne = black; se = white; sw = q2 }`





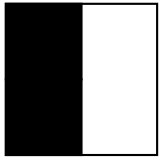
Images par arbres quaternaires - Exercice 1

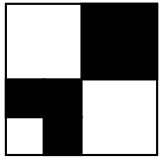
```
type color = Black | White
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Dessiner les images représentés par les valeurs des variables ci-dessous.

1. `let q1 = Square { nw = black; ne = white; se = white; sw = black }`
2. `let q2 = Square { nw = black; ne = black; se = black; sw = white }`
3. `let q3 = Square { nw = white; ne = black; se = white; sw = q2 }`



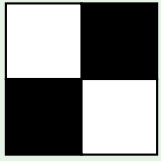


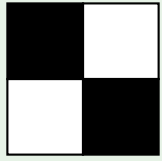
Images par arbres quaternaires - Exercice 2

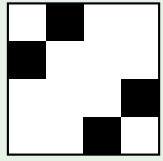
```
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Donner les valeurs de type quad qui représentent les images dessinées ci-dessous.





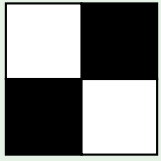


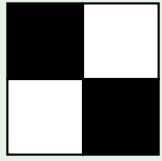
Images par arbres quaternaires - Exercice 2

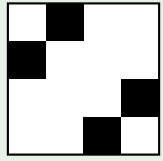
```
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Donner les valeurs de type quad qui représentent les images dessinées ci-dessous.







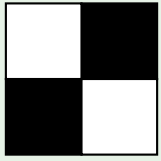
1. `let q3 = Square { nw = white; ne = black; se = white; sw = black }`

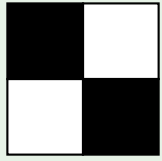
Images par arbres quaternaires - Exercice 2

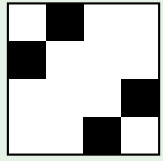
```
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Donner les valeurs de type quad qui représentent les images dessinées ci-dessous.







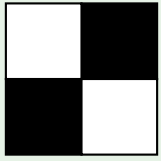
1. `let q3 = Square { nw = white; ne = black; se = white; sw = black }`
2. `let q4 = Square { nw = black; ne = white; se = black; sw = white }`

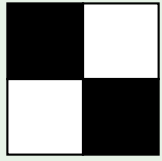
Images par arbres quaternaires - Exercice 2

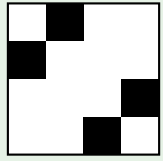
```
type quad =
    Uniform of color
  | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
let black = Uniform Black ;; let white = Uniform White
```

Exercice

Donner les valeurs de type quad qui représentent les images dessinées ci-dessous.







1. `let q3 = Square { nw = white; ne = black; se = white; sw = black }`
2. `let q4 = Square { nw = black; ne = white; se = black; sw = white }`
3. `let q5 = Square { nw = q3; ne = white; se = q3; sw = white }`

Images par arbres quaternaires - Exercice 3 : damier

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```

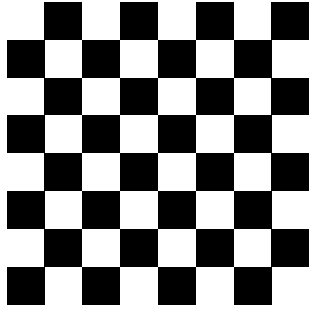
Exercice

Écrire une fonction

```
val checkers : int -> quad
```

telle que `checkers n` calcule une image représentant un plateau de dame de largeur et hauteur 2^n.

Par exemple, `checkers 3` calcule l'image suivante.



Images par arbres quaternaires - Exercice 3 : damier

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```

Exercice

Écrire une fonction

```
val checkers : int -> quad
```

telle que `checkers n` calcule une image représentant un plateau de dame de largeur et hauteur 2^n.

```
let rec checkers n =
  if n = 1 then
    Square { nw = white; ne = black; se = white; sw = black }
  else if n > 1 then
    let small = checkers (n-1) in
    Square { nw = small; ne = small; se = small; sw = small }
  else
    (* ... *)
```

Images par arbres quaternaires - Exercice 3 : damier

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }

let rec checkers n =
  if n = 1 then
    Square { nw = white; ne = black; se = white; sw = black }
  else if n > 1 then
    let small = checkers (n-1) in
    Square { nw = small; ne = small; se = small; sw = small }
  else black
```

Exercice

Comment varie, en fonction de `n`, l'espace occupé par la valeur de `checkers n` ? Et le temps de calcul de cette valeur ?

Images par arbres quaternaires - Exercice 3 : damier

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }

let rec checkers n =
  if n = 1 then
    Square { nw = white; ne = black; se = white; sw = black }
  else if n > 1 then
    let small = checkers (n-1) in
    Square { nw = small; ne = small; se = small; sw = small }
  else black
```

Exercice

Comment varie, en fonction de `n`, l'espace occupé par la valeur de `checkers n` ? Et le temps de calcul de cette valeur ?

- l'espace occupé et le temps de calcul sont proportionnels à n.

Images par arbres quaternaires - Exercice 3 : damier

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }

let rec checkers n =
  if n = 1 then
    Square { nw = white; ne = black; se = white; sw = black }
  else if n > 1 then
    Square { nw = checkers (n-1); ne = checkers (n-1);
             se = checkers (n-1); sw = checkers (n-1) }
  else black
```

Exercice

Comment varie, en fonction de `n`, l'espace occupé par la valeur de `checkers n` ? Et le temps de calcul de cette valeur ?

Images par arbres quaternaires - Exercice 3 : damier

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }

let rec checkers n =
  if n = 1 then
    Square { nw = white; ne = black; se = white; sw = black }
  else if n > 1 then
    Square { nw = checkers (n-1); ne = checkers (n-1);
             se = checkers (n-1); sw = checkers (n-1) }
  else black
```

Exercice

Comment varie, en fonction de `n`, l'espace occupé par la valeur de `checkers n` ? Et le temps de calcul de cette valeur ?

- l'espace occupé et le temps de calcul sont proportionnels à 4^n.

Images par arbres quaternaires - Exercice 4 : opérations

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```

Exercice

Écrire les fonctions suivantes

1. `vertical_mirror : quad -> quad` (symétrie gauche-droite)
2. `point_reflection : quad -> quad` (symétrie centrale)
3. `union : quad -> quad -> quad` (blanc = transparent)

Images par arbres quaternaires - Exercice 4 : opérations

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```

Exercice

Écrire les fonctions suivantes

1. `vertical_mirror : quad -> quad` (symétrie gauche-droite)
2. `point_reflection : quad -> quad` (symétrie centrale)
3. `union : quad -> quad -> quad` (blanc = transparent)

```
let rec vertical_mirror = function
  | Uniform c -> Uniform c
  | Square { nw=nw; ne=ne; se=se; sw=sw } ->
      Square { nw=vertical_mirror ne; ne=vertical_mirror nw;
               sw=vertical_mirror se; se=vertical_mirror sw }
```

Images par arbres quaternaires - Exercice 4 : opérations

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```

Exercice

Écrire les fonctions suivantes

1. `vertical_mirror : quad -> quad` (symétrie gauche-droite)
2. `point_reflection : quad -> quad` (symétrie centrale)
3. `union : quad -> quad -> quad` (blanc = transparent)

```
let rec point_reflection = function
  | Uniform c -> Uniform c
  | Square { nw=nw; ne=ne; se=se; sw=sw } ->
      Square { nw=point_reflection se; ne=point_reflection sw;
               sw=point_reflection ne; se=point_reflection nw }
```

Images par arbres quaternaires - Exercice 4 : opérations

```
type quad = Uniform of color
          | Square of { nw : quad; ne : quad; se : quad; sw : quad; }
```

Exercice

Écrire les fonctions suivantes

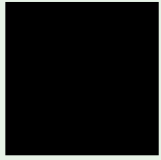
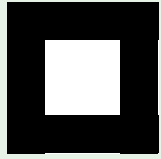
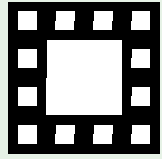
1. `vertical_mirror : quad -> quad` (symétrie gauche-droite)
2. `point_reflection : quad -> quad` (symétrie centrale)
3. `union : quad -> quad -> quad` (blanc = transparent)

```
let rec union s1 s2 = match (s1, s2) with
  | (Uniform Black, _) | (_, Uniform Black) -> Uniform Black
  | (Uniform White, s) | (s, Uniform White) -> s
  | (Square { nw=nw1; ne=ne1; se=se1; sw=sw1 },
     Square { nw=nw2; ne=ne2; se=se2; sw=sw2 }) ->
       Square { nw=union nw1 nw2; ne=union ne1 ne2;
                sw=union sw1 sw2; se=union se1 se2 }
```

Images par arbres quaternaires - Exercice 5 : Sierpińki

Exercice

Écrire une fonction `fractal : int -> quad` qui construit un arbre quaternaire en itérant le processus dont les trois premières étapes sont les suivantes.

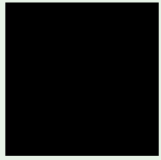
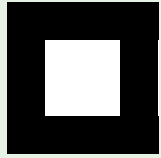
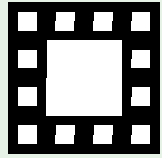




Indication méthodologique : quel est le processus qui permet de passer du résultat de l'itération `n` à celui de l'itération `n + 1` ?

Images par arbres quaternaires - Exercice 5 : Sierpiński

Exercice

Écrire une fonction `fractal : int -> quad` qui construit un arbre quaternaire en itérant le processus dont les trois premières étapes sont les suivantes.

```
let rec fractal n =
  if n <= 0 then Uniform Black
  else
    let small = fractal (n-1) in
    Square { nw = Square { nw=small; ne=small; se=white; sw=small };
             ne = Square { nw=small; ne=small; se=small; sw=white };
             se = Square { nw=white; ne=small; se=small; sw=small };
             sw = Square { nw=small; ne=white; se=small; sw=small }; }
```