CAML

7 - Piles et Files

http://tsi.tuxfamily.org/OCaml

7 avril 2023

On a rencontré dans les chapitres précédents divers types de données prédéfinis par OCaml (`int`, `float`, `bool`, `vect` et `list`). Il est parfois souhaitable, pour résoudre un problème particulier, de définir de nouveaux types de données.

On a rencontré dans les chapitres précédents divers types de données prédéfinis par OCaml (`int`, `float`, `bool`, `vect` et `list`). Il est parfois souhaitable, pour résoudre un problème particulier, de définir de nouveaux types de données.

Vocabulaire

Un **type produit** ou **type enregistrement** est formé d'une liste de rubriques (appelées **champs**) ayant chacune un nom (appelé **étiquette** ou **label**), ce qui permet de réaliser facilement le filtrage, les étiquettes jouant le rôle de motifs.

- Déclaration du type :

```
type personne = {nom : string;
                 prenom : string ;
                    age : int };;
```

- Déclaration d'une variable de ce type :

```
# let p = {nom = "Rotich"; prenom =
                          "Juliana"; age = 43};;
val p : personne = {nom = "Rotich"; prenom =
                            "Juliana"; age = 43}
```

- Déclaration d'une variable de ce type :

```
# let p = {nom = "Rotich"; prenom =
                          "Juliana"; age = 43};;
val p : personne = {nom = "Rotich"; prenom =
                            "Juliana"; age = 43}
```

J. Rotich, née en 1977 au Kenya, est une professionnelle des technologies informatiques qui a notamment participé à la création du premier logiciel open-source ≪ made in Africa ≫, **Ushahidi**, pour cartographier les dégâts et les témoignages dans les situations de crise, et qui dirige l'association de même nom. Elle travaille également à des solutions techniques facilitant l'accès à Internet sur tout le territoire africain. Source : femmedinfluence.fr

- Lecture d'un champ :

```
# p.nom;;
- : string = "Rotich"
```

- Lecture d'un champ :

```
# p.nom;;
- : string = "Rotich"
```

- Écriture :

```
# p.age <- 53;;
Characters 2-13:
  p.age <- 53;;
  ^^^^^^^^^^^
Error: The record field age is not mutable
```

- Lecture d'un champ :

```
# p.nom;;
- : string = "Rotich"
```

- Écriture :

```
# p.age <- 53;;
Characters 2-13:
  p.age <- 53;;
  ^^^^^^^^^^^
Error: The record field age is not mutable
```

Il faut donc préciser les champs destinés à être modifiés lors de la déclaration du type.

```
# type personne = {nom : string; prenom : string
                   ; mutable age : int};;

# let p = {nom = "Rotich"; prenom = "Juliana";
                                     age = 43};;

val p : personne = {nom = "Rotich"; prenom =
                          "Juliana"; age = 43}
```

```
# type personne = {nom : string; prenom : string
                   ; mutable age : int};;

# let p = {nom = "Rotich"; prenom = "Juliana";
                                     age = 43};;

val p : personne = {nom = "Rotich"; prenom =
                          "Juliana"; age = 43}
```

```
# p.age <- 53;;
- : unit = ()

# p;;
- : personne = {nom = "Rotich"; prenom =
                            "Juliana"; age = 53}
```

Enregistrements polymorphes :

- Déclaration d'une référence :

```
type 'a myref = {mutable valeur : 'a};;
```

Enregistrements polymorphes :

- Déclaration d'une référence :

```
type 'a myref = {mutable valeur : 'a};;
```

- Accéder au contenu (!x) :

```
let contenu x = x.valeur;;
```

Enregistrements polymorphes :

- Déclaration d'une référence :

```
type 'a myref = {mutable valeur : 'a};;
```

- Accéder au contenu (`!x`) :

```
let contenu x = x.valeur;;
```

- Modifier le contenu (`x := 3`) :

```
let affect x a = x.valeur <- a;;
```

- Un exemple :

```
# let x = {valeur=0};;
val x : int myref = {valeur = 0}
# affect x (contenu x +1);;
- : unit = ()
# x.valeur;;
- : int = 1
```

- Un exemple :

```
# let x = {valeur=0};;
val x : int myref = {valeur = 0}
# affect x (contenu x +1);;
- : unit = ()
# x.valeur;;
- : int = 1
```

- Un autre :

```
# let y = {valeur = 2.1};;
val y : float myref = {valeur = 2.1}
# affect y (contenu y +. 0.1);;
- : unit = ()
# y.valeur <- y.valeur +. 0.1;;
- : unit = ()
# y.valeur;;
- : float = 2.3000000000000003
```

Vocabulaire

- Les **variants** sont des types énumérés.

Vocabulaire

- Les **variants** sont des types énumérés.
- Au début de chacune des sections, séparées par des | , se trouve un **constructeur**.

Vocabulaire

- Les **variants** sont des types énumérés.
- Au début de chacune des sections, séparées par des | , se trouve un **constructeur**.
- On peut les nommer comme on veut, tant que leur nom commence par une **majuscule**.

```
# type booleen = Vrai | Faux;;
type booleen = Vrai | Faux
```

```
# type booleen = Vrai | Faux;;
type booleen = Vrai | Faux
```

```
# let et a = function
          | Vrai -> a
          | Faux -> Faux
;;
```

```
# type booleen = Vrai | Faux;;
type booleen = Vrai | Faux
```

```
# let et a = function
          | Vrai -> a
          | Faux -> Faux
;;
```

```
val et : booleen -> booleen -> booleen = <fun>
```

```
# type booleen = Vrai | Faux;;
type booleen = Vrai | Faux
```

```
# let et a = function
          | Vrai -> a
          | Faux -> Faux
;;
```

```
val et : booleen -> booleen -> booleen = <fun>
```

```
# et Faux Faux;;
- : booleen = Faux
```

On cherche à modéliser les cartes d'un paquet de cartes.

Une première solution :

```
type couleur= Pique | Coeur | Carreau | Trefle;;
type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Petite_carte of int * couleur;;
```

Une première solution :

```
type couleur= Pique | Coeur | Carreau | Trefle;;
type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Petite_carte of int * couleur;;
```

```
# As Coeur;;
- : carte = As Coeur
# Petite_carte (7,Pique);;
- : carte = Petite_carte (7, Pique)
# Petite_carte (7,Pique);;
- : carte = Petite_carte (7, Pique)
```

Une deuxième solution : à partir des types :

```
type couleur = Pique|Trefle|Coeur|Carreau ;;
type hauteur = As|Roi|Dame|Valet|Val of int;;
```

On crée le type carte sous forme d'un enregistrement

```
type carte = { coul : couleur; haut : hauteur};

# {coul = Pique ; haut = Val 2};;
- : carte = {coul = Pique; haut = Val 2}
```

Autre exemple, créer une fonction somme sur le type `nombre` :

```
type nombre = Entier of int | Reel of float;
```

Autre exemple, créer une fonction somme sur le type `nombre` :

```
type nombre = Entier of int | Reel of float;
```

```
let somme x y = match (x,y) with
    Entier a, Entier b ->Entier(a+b)
  |Entier a, Reel b ->Reel(float_of_int a +. b)
  |Reel a, Entier b ->Reel(a+.(float_of_int b))
  |Reel a, Reel b ->Reel(a +. b)
;;
```

Autre exemple, créer une fonction somme sur le type `nombre` :

```
type nombre = Entier of int | Reel of float;
```

```
let somme x y = match (x,y) with
    Entier a, Entier b ->Entier(a+b)
  |Entier a, Reel b ->Reel(float_of_int a +. b)
  |Reel a, Entier b ->Reel(a+.(float_of_int b))
  |Reel a, Reel b ->Reel(a +. b)
;;
```

```
val somme : nombre -> nombre -> nombre = <fun>
```

```
# let a = Entier 4 and
      b = Reel 2.3 and
      c = Entier 1;;
val a : nombre = Entier 4
val b : nombre = Reel 2.3
val c : nombre = Entier 1
# somme a b;;
- : nombre = Reel 6.3
# somme b b;;
- : nombre = Reel 4.6
# somme a c;;
- : nombre = Entier 5
```

```
# let a = Entier 4 and
      b = Reel 2.3 and
      c = Entier 1;;
val a : nombre = Entier 4
val b : nombre = Reel 2.3
val c : nombre = Entier 1
# somme a b;;
- : nombre = Reel 6.3
# somme b b;;
- : nombre = Reel 4.6
# somme a c;;
- : nombre = Entier 5
```

C'est d'ailleurs le même fonctionnement en Python, sauf que ces changements de type ont déjà été programmés.

```
>>> a = 4
>>> b = 2.3
>>> c = 1
>>> a + b
6.3
>>> b + b
4.6
>>> a + c
5
>>> 2.3 + 1.7
4.0
```

Vocabulaire

- Une **structure de données** est une collection d'objets qui ont des propriétés identiques indépendantes de toute représentation informatique.

Vocabulaire

- Une **structure de données** est une collection d'objets qui ont des propriétés identiques indépendantes de toute représentation informatique.
- L'ensemble des fonctions de manipulation associées est appelé la **signature** de la structure de données.

Vocabulaire

Une **pile** (stack) est une liste ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée **sommet**(*top*) de la pile :

Vocabulaire

Une **pile** (stack) est une liste ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée **sommet**(*top*) de la pile :

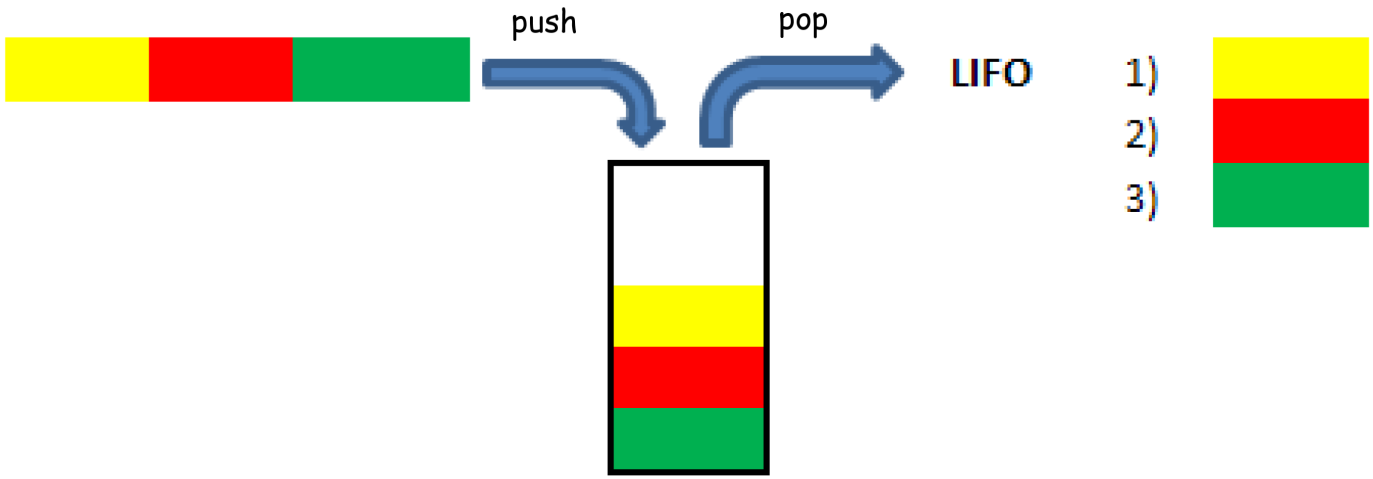
- **Empiler** (*push*) un objet sur une pile P consiste à insérer cet objet au sommet de P.

Vocabulaire

Une **pile** (stack) est une liste ne permettant des insertions ou des suppressions qu'à une seule extrémité, appelée **sommet**(*top*) de la pile :

- **Empiler** (*push*) un objet sur une pile P consiste à insérer cet objet au sommet de P.
- **Dépiler** (*pop*) un objet de P consiste à supprimer de P l'objet placé au sommet. L'objet dépilé est retourné par la fonction de dépilement pour être traité par le programme.

Piles



Processus LIFO (**L**ast **I**n, **F**irst **O**ut)

Pour la pile, on dispose de la bibliothèque Stack et des fonctions create, push et pop

```
# let p = Stack.create();;
val p : '_weak1 Stack.t = <abstr>
# for i = 1 to 5 do Stack.push i p done ;;
- : unit = ()
# p;;
- : int Stack.t = <abstr>
# for i = 1 to 5 do
        print_int(Stack.pop p);
        print_string " "
done;;
5 4 3 2 1 - : unit = ()
# p;;
- : int Stack.t = <abstr>
# Stack.pop p;;
Exception: Stdlib.Stack.Empty.
```

- On dispose aussi de `Stack.is_empty : 'a t -> bool` qui renvoie `true` si la pile donnée est vide et `false` sinon.

- On dispose aussi de `Stack.is_empty : 'a t -> bool` qui renvoie `true` si la pile donnée est vide et `false` sinon.
- Et d'une autre pour connaitre le nombre d'éléments dans la pile : `Stack.length : 'a t -> int` renvoie le nombre d'éléments de la pile donnée. La complexité de cette fonction est en $O(1)$.

Vocabulaire

Pour une **file**, c'est le principe FIFO qui est implémenté. Une file possède une **tête** et une **queue** :

Vocabulaire

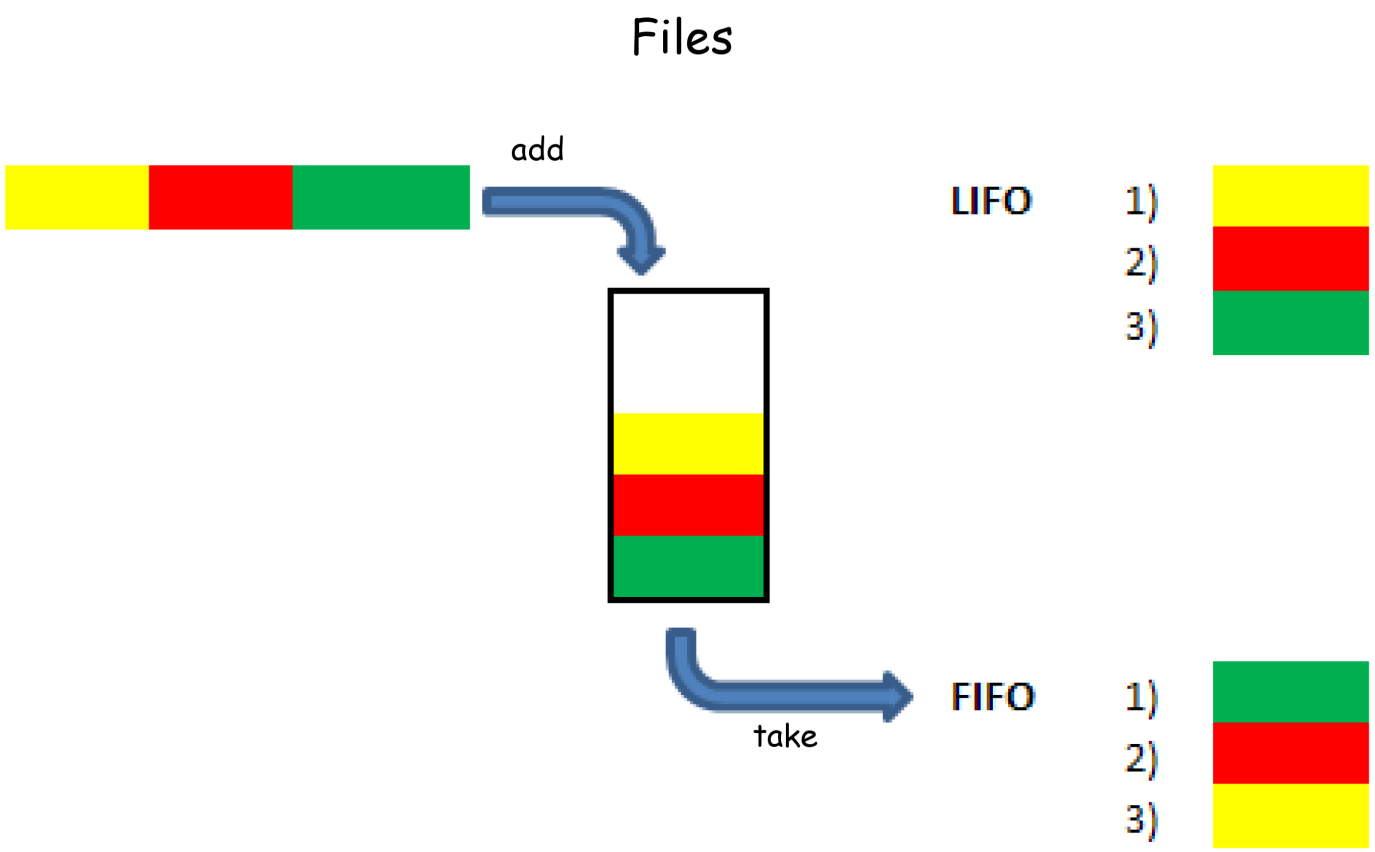
Pour une **file**, c'est le principe FIFO qui est implémenté. Une file possède une **tête** et une **queue** :

- **Enfiler** (*add* ou *push*) un objet sur une file F consiste à insérer cet objet en queue de F.

Vocabulaire

Pour une **file**, c'est le principe FIFO qui est implémenté. Une file possède une **tête** et une **queue** :

- **Enfiler** (*add* ou *push*) un objet sur une file F consiste à insérer cet objet en queue de F.
- **Défiler** (*take* ou *pop*) un objet de F consiste à récupérer et supprimer de F l'objet placé en tête.



Processus FIFO (**F**irst **I**n, **F**irst **O**ut)

Pour la file, on dispose du module Queue et des fonctions create, push et pop

```
# let f = Queue.create();;
val f : '_weak2 Queue.t = <abstr>
# for i = 1 to 5 do Queue.push i f done;;
- : unit = ()
# f;;
- : int Queue.t = <abstr>
# for i = 1 to 5 do
        print_int(Queue.pop f);
        print_string " "
done;;
1 2 3 4 5 - : unit = ()
# f;;
- : int Queue.t = <abstr>
# Queue.pop f;;
Exception: Stdlib.Queue.Empty.
```