OCaml

Prise de contact et traits fonctionnels

Crédits

- Developpez.com
- λ OCaml Programming
- FAQ Caml

Tableaux

Déclaration d'un tableau :

```
let tab = [|3;6;8|];; (*à la main*)

let tab = Array.make 5 0;;(*un tableau de 5 zéros*)

(*Créer une matrice à la main*)
let mat = let p = Array.make 3 [||] in
  p.(0)<-[|1;2|]; p.(1) <-[|0;9|]; p.(2)<-[|-1;1|]; p;;

(*utiliser une fonction de bibliothèque*)
Array.make_matrix 3 2 0;;
```

Accès aux éléments

```
# let tab = [|3;6;8|] in
tab.(0)<-10;
Printf.printf "tab.(%d)=%d\n" 0 tab.(0);;
    tab.(0)=10
- : unit = ()

# let mat = Array.make_matrix 3 2 0 in
mat.(2).(1)<- -4; mat;;
  - : int array array = [|[|0; 0|]; [|0; 0|]; [|0; -4|]|]
```

Parcours d'un tableau, longueur, boucle for

Parcours à l'endroit ; à l'envers :

```
# let tab = [|3;6;8|];;
 val tab : int array = [|3; 6; 8|]

# let n = Array.length tab  in
for i = 0 to (n-1) do (*à l'endroit*)
  Printf.printf "tab.(%d)=%d; " i tab.(i);
done;;
      tab.(0)=3; tab.(1)=6; tab.(2)=8; - : unit = ()

# let n = Array.length tab  in
  for i = n-1 downto 0 do (*à l'envers*)
    Printf.printf "tab.(%d)=%d; " i tab.(i);
  done;;
      tab.(2)=8; tab.(1)=6; tab.(0)=3; - : unit = ()
```

Références

Une référence est un pointeur vers un objet. Cela permet de rendre des variables mutables.

```
  # let n = ref 0;;
val n : int ref = {contents = 0}
# n := !n+3;;
- : unit = ()
# Printf.printf "!n = %d\n" !n;;
!n = 3
- : unit = ()
```

Egalités

- `==` : égalité physique.

```
# [1] == [1];;
- : bool = false
```

Les deux listes ne sont pas au même endroit de la mémoire.

Egalités

- `==` : égalité physique.

```
# [1] == [1];;
- : bool = false
```

Les deux listes ne sont pas au même endroit de la mémoire.

- `=` : égalité syntaxique.

```
# [1] = [1];;
- : bool = true
```

Les deux listes « s'écrivent pareillement ».

Egalités

- `==` : égalité physique.

```
# [1] == [1];;
- : bool = false
```

Les deux listes ne sont pas au même endroit de la mémoire.

- `=` : égalité syntaxique.

```
# [1] = [1];;
- : bool = true
```

Les deux listes « s'écrivent pareillement ».

- `!=` : différence physique.

```
# [1] != [1];;
- : bool = true
```

Egalités

- `==` : égalité physique.

```
# [1] == [1];;
- : bool = false
```

Les deux listes ne sont pas au même endroit de la mémoire.

- `=` : égalité syntaxique.

```
# [1] = [1];;
- : bool = true
```

Les deux listes « s'écrivent pareillement ».

- `!=` : différence physique.

```
# [1] != [1];;
- : bool = true
```

- `<>` : différence syntaxique.

```
# [1] <> [1];;
- : bool = false
```

Boucle while

```
# let i = ref 0 and s = ref 0 in
while !i<10 do
  s:= !s + !i;
  incr i;(*ou i := !i + 1*)
done;
Printf.printf "!s = %d\n" !s;;
           !s = 45
- : unit = ()
```

Noter le `incr i` qui incrémente de 1 la valeur pointée par `i`. Il existe aussi un `decr i`.

Présentation

- En programmation fonctionnelle, les fonctions sont totales, c'est-à-dire qu'elles sont applicables à tout argument qui appartient à leur type de départ.

Présentation

- En programmation fonctionnelle, les fonctions sont totales, c'est-à-dire qu'elles sont applicables à tout argument qui appartient à leur type de départ.
- Il faut donc être capable de traiter les cas, appelés *exceptions* , où cet argument n'est pas acceptable. Par exemple : une division par zéro ou bien la recherche de la tête d'une liste vide.

Présentation

- En programmation fonctionnelle, les fonctions sont totales, c'est-à-dire qu'elles sont applicables à tout argument qui appartient à leur type de départ.
- Il faut donc être capable de traiter les cas, appelés *exceptions* , où cet argument n'est pas acceptable. Par exemple : une division par zéro ou bien la recherche de la tête d'une liste vide.
- Ceci peut être fait par des tests préventifs placés dans le corps des fonctions, mais ce mécanisme est très lourd, car il implique un travail important pour le programmeur et il altère la lisibilité d'un programme en masquant son fonctionnement normal.

Présentation

- En programmation fonctionnelle, les fonctions sont totales, c'est-à-dire qu'elles sont applicables à tout argument qui appartient à leur type de départ.
- Il faut donc être capable de traiter les cas, appelés *exceptions* , où cet argument n'est pas acceptable. Par exemple : une division par zéro ou bien la recherche de la tête d'une liste vide.
- Ceci peut être fait par des tests préventifs placés dans le corps des fonctions, mais ce mécanisme est très lourd, car il implique un travail important pour le programmeur et il altère la lisibilité d'un programme en masquant son fonctionnement normal.
- C'est pourquoi les langages de programmation modernes comportent un mécanisme spécifique pour le traitement des exceptions.

Exceptions prédéfinies

Les exceptions ont le type `exn`. On les soulève avec la commande `raise`.

- Une exception qui porte bien son nom :

```
# Exit;;
- : exn = Pervasives.Exit
# raise Exit;;
Exception: Pervasives.Exit.
```

Exceptions prédéfinies

Les exceptions ont le type `exn`. On les soulève avec la commande `raise`.

- Une exception qui porte bien son nom :

```
# Exit;;
- : exn = Pervasives.Exit
# raise Exit;;
Exception: Pervasives.Exit.
```

- Une qu'on connaît bien

```
# failwith "big pb !";;
Exception: Failure "big pb !".
```

Observons que l'invocation de `failwith` déclenche le `raise`.

Exceptions prédéfinies

Les exceptions ont le type `exn`. On les soulève avec la commande `raise`.

- Une exception au sens compréhensible

```
1 # 1/0;;
2 Exception: Division_by_zero.
```

C'est l'évaluateur de OCaml qui déclenche cette exception.

Exceptions prédéfinies

Les exceptions ont le type `exn`. On les soulève avec la commande `raise`.

- Une exception au sens compréhensible

```
# 1/0;;
Exception: Division_by_zero.
```

C'est l'évaluateur de OCaml qui déclenche cette exception.

- Une autre qui rappelle le C ou encore Python.

```
 # let l = [3] in assert (List.mem 2 l);;
Exception: Assert_failure ("//toplevel//", 1, 15).
```

Déclarer une exception

```
# exception MyException;;
exception MyException
# MyException;;
- : exn = MyException
# raise MyException;;
Exception: MyException.
# exception MyException2 of string;;
exception MyException2 of string
# raise (MyException2 "big pb");;
Exception: MyException2 "big pb".
```

Déclarer une exception

```
# exception MyException2 of string;;
exception MyException2 of string
# let f = function x ->
  if x <> 0 then
    365 / x
  else
    raise (MyException2 "Division par zéro");;
        val f : int -> int = <fun>
# f 3;;
- : int = 121
# f 0;;
Exception: MyException2 "Division par zéro".
```

Récupération d'une exception

La récupération d'une exception déclenchée lors de l'évaluation d'une expression `e` peut être réalisée en encapsulant cette expression dans une expression `try..with`.
Voici la syntaxe à utiliser :

```
 try expr with
| p1 -> e1 (*si le type d'exception est p1, renvoyer e1*)
| ...
| pn -> en     (*si le type d'exception est p1, renvoyer e1*)
```

Dans l'exemple suivant, on met la tête de la liste `l2` dans `l1` et, si une exception de type `Failure` est soulevée, on renvoie la liste vide :

```
# let ajouter l1 l2 =
  try List.hd l2::l1 with Failure _ -> [];;
  val ajouter : 'a list -> 'a list -> 'a list = <fun>
# ajouter [2] [3;4];;
- : int list = [3; 2]
# ajouter [2] [];;
- : int list = []
```

Présentation

L'idée est celle des `struct` de C. On accède aux champs avec la notation pointée ; on les modifie avec la notation fléchée des tableaux.

```
# type complex = {x:float; y:float};;
type complex = { x : float; y : float; }
# let i = {x=0.; y=1.};;
val i : complex = {x = 0.; y = 1.}
# i.x, i.y;;
- : float * float = (0., 1.)
# i.x=4.;;
- : bool = false
# i.x<-4.;;
Characters 0-7:
  i.x<-4.;;
  ^^^^^^^
Error: The record field x is not mutable
```

Champs mutables

Il peut ête souhaitable de modifier certains champs. Il faut les déclarer comme `mutable`. Par défaut, un champ est persistant (ou immutable).

```
# type complex = {mutable x:float; y:float};;
type complex = { mutable x : float; y : float; }
# let i = {x=1.; y=1.};;
val i : complex = {x = 0.; y = 1.}
# i.x, i.y;;
- : float * float = (0., 1.)
# i.x=4.;;
- : bool = false
# i.x<-4.;;
- : unit = ()
# i.y<-1.;;
Characters 0-7:
  i.y<-1.;;
  ^^^^^^^
Error: The record field y is not mutable
```

Polymorphisme

Dans l'exemple ci-dessous, on déclare une structure à deux champs `x,y` dont les types ne sont pas connus au départ. Le premier est d'un certain type `'a` qui sera connu à l'initialisation, de même le second est d'un type `'b` .

```
# type ('a,'b) fourre_tout = {x: 'a; y: 'b};;
type ('a, 'b) fourre_tout = { x : 'a; y : 'b; }
# let z = {x=2; y="toto"};;
val z : (int, string) fourre_tout = {x = 2; y = "toto"}
```

Présentation

- Un *type somme* est formé d'une liste de cas possibles pour une valeur de ce type, chaque cas comporte un nom de cas, le « constructeur », et une (éventuelle) valeur associée (l'argument du constructeur).

Présentation

- Un *type somme* est formé d'une liste de cas possibles pour une valeur de ce type, chaque cas comporte un nom de cas, le « constructeur », et une (éventuelle) valeur associée (l'argument du constructeur).
- Un cas dégénéré consiste à définir un type dont les constructeurs n'ont pas d'argument (constructeurs constants). On parle alors de *type énuméré* (le symbole `|` se lit « ou ») :

```
#type couleur = Bleu | Blanc | Rouge;;
type couleur = Bleu | Blanc | Rouge
#Bleu;;
- : couleur = Bleu
# let fleur c = match c with
   | Bleu -> "bleuet"
   | Blanc -> "marguerite"
   | Rouge -> "coquelicot"
in fleur Rouge;;
          - : string = "coquelicot"
```

Type somme

On peut passer des paramètres aux constructeurs.

- Ci-dessous on définit un type pour les piles d'entiers. Une pile non vide posséde deux éléments : une étiquette entière et une pile (vide éventuellement). On écrit une fonction qui fait la somme du contenu de la liste :

```
# type stack_of_int = Vide | P of int * stack_of_int;;
type stack_of_int = Vide | P of int * stack_of_int
# let rec somme p = match p with
  | Vide -> 0
  | P (x,q) -> x + somme q;;
    val somme : stack_of_int -> int = <fun>
```

Type somme

On peut passer des paramètres aux constructeurs.

- Ci-dessous on définit un type pour les piles d'entiers. Une pile non vide posséde deux éléments : une étiquette entière et une pile (vide éventuellement). On écrit une fonction qui fait la somme du contenu de la liste :

```
# type stack_of_int = Vide | P of int * stack_of_int;;
type stack_of_int = Vide | P of int * stack_of_int
# let rec somme p = match p with
   | Vide -> 0
   | P (x,q) -> x + somme q;;
     val somme : stack_of_int -> int = <fun>
```

- On crée ensuite une pile dont le sommet est 3, l'élément intermédiaire 2 et la base 1. On lui applique la fonction `somme` :

```
# let p = P(3,P(2,P(1,Vide)));;
val p : stack_of_int = P (3, P (2, P (1, Vide)))
# somme p;;
- : int = 6
```

Type somme polymorphe

- Ci-dessous on définit un type pour les piles polymorphes. On écrit une fonction qui prend en paramètre une pile, une addition adaptée et une valeur de départ.

```
# type 'a stack = Vide | P of 'a * 'a stack;;
type 'a stack = Vide | P of 'a * 'a stack
# let rec somme_polymorphe p add start = match p with
  | Vide -> start
  | P (x,q) -> add x (somme_polymorphe q add start);;
    val somme_polymorphe : 'a stack -> ('a -> 'b -> 'b)
        -> 'b -> 'b = <fun>
```

Type somme polymorphe

- Ci-dessous on définit un type pour les piles polymorphes. On écrit une fonction qui prend en paramètre une pile, une addition adaptée et une valeur de départ.

```
# type 'a stack = Vide | P of 'a * 'a stack;;
type 'a stack = Vide | P of 'a * 'a stack
# let rec somme_polymorphe p add start = match p with
  | Vide -> start
  | P (x,q) -> add x (somme_polymorphe q add start);;
    val somme_polymorphe : 'a stack -> ('a -> 'b -> 'b)
        -> 'b -> 'b = <fun>
```

- On crée ensuite une pile de flottant. On lui applique la fonction `somme_polymorphe` à laquelle on passe l'addition des flottants `(+.)` et le point de départ `0.` :

```
# let p = P(0.3,P(0.2,P(0.1,Vide)));;
val p : float stack = P (0.3, P (0.2, P (0.1, Vide)))
# somme_polymorphe p (+.) 0.;;
- : float = 0.60000000000000089
```

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.
- Par exemple, la fonction `list_max` renvoie le maximum d'une liste si elle n'est pas vide. Mais on ne sait trop quoi faire avec la liste vide :

```
1 let rec list_max = function
2   | [] -> ???
3   | h :: t -> max h (list_max t)
```

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.
- Par exemple, la fonction `list_max` renvoie le maximum d'une liste si elle n'est pas vide. Mais on ne sait trop quoi faire avec la liste vide :

```
1 let rec list_max = function
2   | [] -> ???
3   | h :: t -> max h (list_max t)
```

- Pour la liste vide, on peut :

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.
- Par exemple, la fonction `list_max` renvoie le maximum d'une liste si elle n'est pas vide. Mais on ne sait trop quoi faire avec la liste vide :

```
1 let rec list_max = function
2   | [] -> ???
3   | h :: t -> max h (list_max t)
```

- Pour la liste vide, on peut :
  - renvoyer une `min_int` ? mais le code ne fonctionnerait qu'avec une liste d'entiers

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.
- Par exemple, la fonction `list_max` renvoie le maximum d'une liste si elle n'est pas vide. Mais on ne sait trop quoi faire avec la liste vide :

```
1 let rec list_max = function
2   | [] -> ???
3   | h :: t -> max h (list_max t)
```

- Pour la liste vide, on peut :
  - renvoyer une `min_int` ? mais le code ne fonctionnerait qu'avec une liste d'entiers
  - Soulever une exception ? mais il faudra que l'utilisateur se souvienne d'encapsuler ses appels dans un `try..with`

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.
- Par exemple, la fonction `list_max` renvoie le maximum d'une liste si elle n'est pas vide. Mais on ne sait trop quoi faire avec la liste vide :

```
1 let rec list_max = function
2   | [] -> ???
3   | h :: t -> max h (list_max t)
```

- Pour la liste vide, on peut :
  - renvoyer une `min_int` ? mais le code ne fonctionnerait qu'avec une liste d'entiers
  - Soulever une exception ? mais il faudra que l'utilisateur se souvienne d'encapsuler ses appels dans un `try..with`
  - Renvoyer `NULL` ? mais cette notion existe en C pas en OCaml

Utilité du type option

- Nous voulons écrire une fonction qui retourne en général une valeur mais, parfois, ne retourne rien.
- Par exemple, la fonction `list_max` renvoie le maximum d'une liste si elle n'est pas vide. Mais on ne sait trop quoi faire avec la liste vide :

```
1 let rec list_max = function
2   | [] -> ???
3   | h :: t -> max h (list_max t)
```

- Pour la liste vide, on peut :
  - renvoyer une `min_int` ? mais le code ne fonctionnerait qu'avec une liste d'entiers
  - Soulever une exception ? mais il faudra que l'utilisateur se souvienne d'encapsuler ses appels dans un `try..with`
  - Renvoyer `NULL` ? mais cette notion existe en C pas en OCaml
- La meilleure solution est d'employer le type `option`.

Le type option

- Le type `'a option` possède deux constructeurs `Some` et `None`.

Le type option

- Le type `'a option` possède deux constructeurs `Some` et `None`.
- un élément du type `option` peut être vu comme une boîte qui est soit vide, soit contenant un objet d'un certain type.

```
# None;;
- : 'a option = None
# Some 45;;
- : int option = Some 45
# Some "toto";;
- : string option = Some "toto"
```

Type option : Accès au contenu

On accède au contenu de la boîte par filtrage. Ci-dessous on écrit une fonction qui extrait un entier d'une option (s'il y en a un dedans) et le convertit en `string` :

```
# let extract o =
  match o with
  | Some i -> string_of_int i
  | None -> "";;
      val extract : int option -> string = <fun>
# extract (Some 42);;
- : string = "42"
# extract None;;
- : string = ""
```

Maximum d'une liste

On revient au programme de recherche du maximum. On ne renvoie rien (donc `None`) dans le cas où la liste est vide.

```
# let rec list_max = function
  | [] -> None
  | h :: t -> begin
      match list_max t with
        | None -> Some h
        | Some m -> Some (max h m)
    end;;
              val list_max : 'a list -> 'a option = <fun>
# list_max [5;1;9;2];;
- : int option = Some 9
# list_max [];;
- : 'a option = None
```