

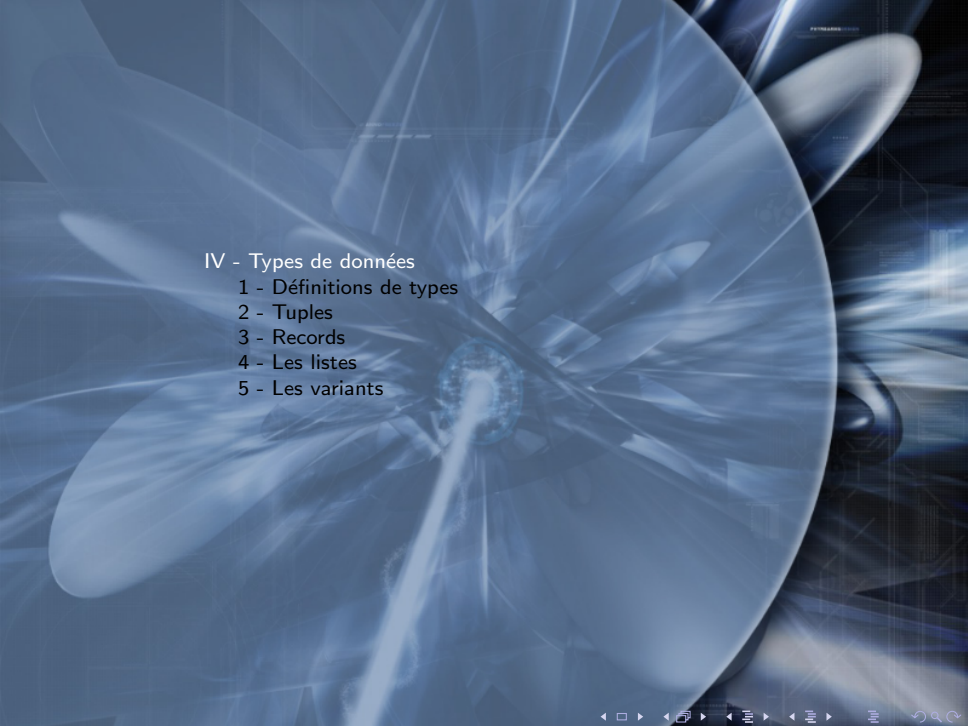
# Programmation fonctionnelle

## Avec Objective Caml

David "Thor" GIRON et l'équipe Koala OCaml

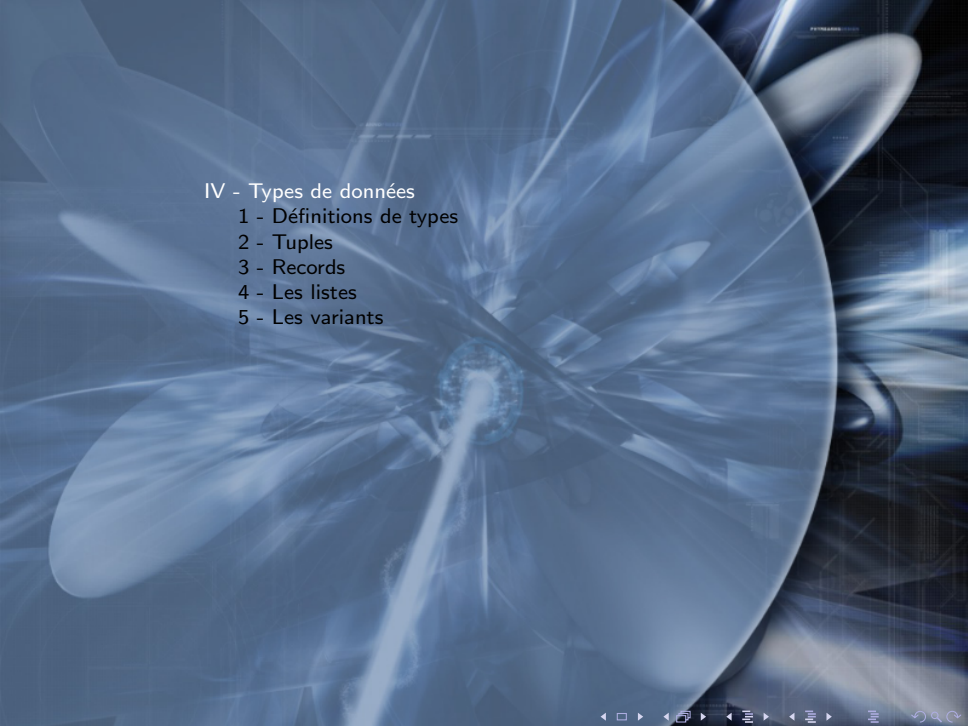
Mardi, 12/10/2010





## IV - Types de données

- 1 - Définitions de types
- 2 - Tuples
- 3 - Records
- 4 - Les listes
- 5 - Les variants



## IV - Types de données

- 1 - Définitions de types
- 2 - Tuples
- 3 - Records
- 4 - Les listes
- 5 - Les variants

# Syntaxe

Il est possible en OCaml de définir de nouveaux types a la manière des typedefs du C.

Exemple :

```
1  # type size = int;;  
2  type size = int  
3  
4  # type grade = char;;  
5  type grade = char
```



Content de vous revoir les gars. Gros  
morceau au menu aujourd'hui.



## Types paramétrés

Il est également possible de paramétrer un type avec un ou plusieurs types.

Exemple :

```
1 # type 'a alias = 'a;;
2 type 'a alias = 'a
```

Cela signifie que le type "alpha alias" est un type alpha, par exemple, le type "int alias" est le type int, le type "string alias" est le type string, etc.



Nous reviendrons sur cette syntaxe lorsque nous aurons des types de données plus pertinents. Pour l'instant, ça n'a pas l'air très utile vu d'ici...



## Syntaxe et typage

Un tuple est une structure de données composée d'au moins deux éléments ordonnés pouvant être de types différents.

La syntaxe est la suivante :

```
1 (elem_1, elem_2, ..., elem_n)
```

Et son type est :

```
1 (type_1 * type_2 * ... * type_n)
```

Le type se lit "type\_1 croix type\_2 croix type\_n".



On peut omettre les parenthèses pour le type, mais c'est une pratique que je ne recommande pas car cela rend le code moins lisible.



## Exemples

```
1  # type personne = (string * int);;
2  type personne = string * int
3  # let moi = ("David", 2009);;
4  val moi : string * int = ("David", 2009)
```

On constate que la valeur "moi" est de type "(string \* int)" et non pas de type "personne" car le compilateur essaie toujours d'inférer les types les plus généraux possible.

Nous verrons plus tard dans le cours comment restreindre ce comportement à un type précis.

```
1  # let div_entiers x y = (x/y, x mod y);;
2  val div_entiers : int -> int -> int * int = <fun>
3  # div_entiers 4 2;;
4  - : int * int = (2, 0)
5  # div_entiers 5 2;;
6  - : int * int = (2, 1)
```



## Déconstruction

```

1  # let moi = ("David", 2009);;
2  val moi : string * int = ("David", 2009)
3
4  # let get_nom (nom, promo) = nom;;
5  val get_nom : 'a * 'b -> 'a = <fun>
6  # let get_promo (nom, promo) = promo;;
7  val get_promo : 'a * 'b -> 'b = <fun>
8
9  # get_nom moi;;
10 - : string = "David"
11 # get_promo moi;;
12 - : int = 2009
    
```



Les fonctions sont polymorphiques car rien ne permet au compilateur d'inférer les types des éléments du tuple dans cet exemple.





## Tuples et filtrage

La déconstruction à la volée, par exemple appliquée à un tuple nous permet de faire des filtrages plus puissants :

Exemple :

```

1  # let is_duo = function
2  | ("Tintin", "Milou") -> true
3  | ("Laurel", "Hardy") -> true
4  | ("Jeanne", "Serge") -> true
5  | ("Boule", "Bill") -> true
6  | ("Olivier", "Tom") -> true
7  | _ -> false
8
9  ;;
9  val is_duo : string * string -> bool = <fun>
10
11  # let duo = ("Tintin", "Bill");;
12  val duo : string * string = ("Tintin", "Bill")
13
14  # is_duo duo;;
15  - : bool = false
    
```



Ha... Nostalgie...

## Tuples paramétrés

Tout type peut être paramétré, donc les tuples aussi. Exemple :

```
1 # type 'a assoc = (string * 'a);;
2 type 'a assoc = string * 'a
```

```
1 # type ('a, 'b) paire = ('a * 'b);;
2 type ('a, 'b) paire = 'a * 'b
```



Là, ça devient intéressant.

## Résumé



Les tuples permettent une équivalence avec les structures du C, à la différence qu'il n'est pas possible de nommer les champs d'un tuple : On retient l'ordre des champs à la place.

Voyons à présent une structure de données légèrement plus complexe et plus proche des structures C : les records.

## Syntaxe et typage

Type :

```
1 {champs_1 : type_1; champs_2 : type_2; ...; champs_n : type_t}
```

Syntaxe :

```
1 {champs_1 = value_1; champs_2 = value_2; ...; champs_n = value_n}
```



Etant donné que les champs sont nommés, il n'est pas possible de déclarer des records anonymes.



## Exemples

```
1  # type personne = {nom : string; promo : int};;
2  type personne = { nom : string; promo : int; }
3  # let moi = {nom = "David"; promo = 2009};;
4  val moi : personne = {nom = "David"; promo = 2009}
```

```
1  # type quotient = {resultat : int; reste : int};;
2  type quotient = { resultat : int; reste : int; }
3  # let div_entiers x y = {resultat = x/y; reste = x mod y};;
4  val div_entiers : int -> int -> quotient = <fun>
5  # div_entiers 4 2;;
6  - : quotient = {resultat = 2; reste = 0}
7  # div_entiers 5 2;;
8  - : quotient = {resultat = 2; reste = 1}
```



Le seul type record contenant exactement les champs "resultat" et "reste" étant le type "quotient", le compilateur arrive à inferer ce type.

## Résumé



Les records permettent une équivalence forte avec les structures du C. Toutefois, on leur préférera généralement les tuples, sauf dans certains cas particuliers que nous verrons dans le 4ème cours abordant les traits impératifs d'OCaml.

## Syntaxe et typage

Ocaml propose nativement une structure de type liste. Ce type est paramétré par un autre type qui sera le type des éléments de la liste. Tous les éléments doivent donc être du même type.

Syntaxe :

1 | `[elem_1; elem_2; ...; elem_n]`

Typage :

1 | `'a list`



Cette structure de donnée est probablement celle que vous utiliserez le plus. La bibliothèque standard contient une foule de fonctions pour la manipuler.



## Opérateur d'ajout à une liste

- ▶ La liste vide se note "[]"
- ▶ L'opérateur d'ajout à une liste est " : " (infixe)
- ▶ Son type est donc "'a -> 'a list -> 'a list"

Exemples :

```

1  # [];;
2  - : 'a list = []
3  # 1::2::3::4::[];;
4  - : int list = [1; 2; 3; 4]
5  # let l = [1; 2; 3; 4];;
6  val l : int list = [1; 2; 3; 4]
7  # 0::l;;
8  - : int list = [0; 1; 2; 3; 4]
```





## Opérateur de concatenation

- L'opérateur de concatenation de listes est "@" (infixe)
- Son type est donc "'a list -> 'a list -> 'a list"

Exemples :

```
1  # let l = [1; 2; 3; 4];;
2  val l : int list = [1; 2; 3; 4]
3  # [0]@l;;                                (* Mal ! *)
4  - : int list = [0; 1; 2; 3; 4]
5  # [1; 2; 3; 4]@[5; 6; 7; 8];;
6  - : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```



Ne jamais utiliser @ pour concaténer une liste à 1 seul élément, JAMAIS !



## Parcours d'une liste

Lorsqu'on doit parcourir une liste chaînée en C, la méthode la plus pratique est souvent la récursion avec un algorithme semblable à :

```
1  Parcours(liste)
2      Si liste != vide
3          tete = Premier_element(liste)
4          reste = Suite_liste(liste)
5          Traiter(tete)
6          Parcours(reste)
7      Sinon
8          Return
```



Haaaa... Les parcours de listes, ca me rappelle ma tech1, pas vous ?



## Parcours d'une liste

En Ocaml, en combinant la déconstruction à la volée et le filtrage, parcourir une liste est d'une simplicité déconcertante :

```

1  # let fruits = ["framboise"; "kiwi"; "orange"];;
2  val fruits : string list = ["framboise"; "kiwi"; "orange"]
3
4  # let rec print_list liste = match liste with
5    | [] -> print_endline "Parcours termine !"
6    | tete::reste ->
7      begin
8        print_endline tete;
9        print_list reste
10     end
11  ;;
12  val print_list : string list -> unit = <fun>
13
14  # print_list fruits;;
15  framboise
16  kiwi
17  orange
18  Parcours termine !
19  - : unit = ()
    
```



## Résumé



Les listes sont vraiment le type de données le plus répandu en OCaml. Vous allez en utiliser énormément ! Il est très important de bien maîtriser les quelques lignes de code du slide précédent, c'est pourquoi le premier projet du module concernera la manipulation des listes.

## Les enums en C

En C, il existe les types énumérés (enum) qui permettent de lister toutes les valeurs possibles pour un type.

Exemple :

```
1 | enum jours { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE }
```

Les valeurs LUNDI, MARDI, etc, s'évaluent vers des entiers à partir de 0, mais il est également possible d'explicitier ces entiers quand nécessaire :

```
1 | enum modes { RDONLY = 1, WRONLY = 2, RW = 3 }
```

## Limite des enums en C

Malheureusement il est impossible d'évaluer les valeurs d'un enum vers autre chose que des entiers. Le code suivant est impossible en C :

```
1 enum pseudos
2 {
3     DAVID = "Thor",
4     SULLIVAN = "Sully",
5     DAN = "Danou",
6     MAXIME = "Zaz",
7     URIEL = "Korfuri"
8 }
```



Cela pourrait pourtant s'avérer très pratique !

## Les variants

OCaml possède également un type énuméré, utilisable de la même manière qu'en C.

Syntaxe :

```
1 | type nom_variant = Value_1 | Value_2 | ... | Value_n
```

Chaque valeur "Value\_1", "Value\_2", ..., "Value\_n" est appelé un "constructeur" et doit toujours commencer par une majuscule.

Typage :

```
1 | nom_variant
```



## Les variants

Exemple :

```
1  # type jours = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi |  
   Dimanche;;  
2  type jours = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi |  
   Dimanche  
3  # Lundi;;  
4  - : jours = Lundi  
5  # let week_end = [Samedi; Dimanche];;  
6  val week_end : jours list = [Samedi; Dimanche]
```



Les valeurs Lundi, Mardi, ... s'évaluent vers elles-mêmes, pas vers des entiers comme en C.





## Les variants

Comportement inexistant en C, il est possible de spécifier un type pour chaque constructeur d'un variant avec le mot-clef `of`. Exemple :

```

1  # type exemple =
2    Paire of (int * int)
3    | Mot of string
4    | Autre of exemple;;
5  type exemple =
6    Paire of (int * int)
7    | Mot of string
8    | Autre of exemple
9
10 # let a = Mot "salut";;
11 val a : exemple = Mot "salut"
12 # let b = Paire (42, 32);;
13 val b : exemple = Paire (42, 32)
14 # let c = Autre (Mot "coucou");;
15 val c : exemple = Autre (Mot "coucou")
    
```



Les variants sont récursifs sans mot-clef additionnel.

## Exemple de l'eval\_expr

Armé des variants, du filtrage et de la récursion, coder un eval\_expr est d'une simplicité déroutante :

```

1  # type expression =
2      | Produit of (expression * expression)
3      | Somme of (expression * expression)
4      | Immédiat of int
5
6  ;;
7  type expression =
8      | Produit of (expression * expression)
9      | Somme of (expression * expression)
10     | Immédiat of int
11
12 # let e = Produit (Somme (Immédiat 14, Immédiat 43), Immédiat 2);;
13 val e : expression = Produit (Somme (Immédiat 14, Immédiat 43), Immédiat 2)
14
15 # let rec eval_expr = function
16     | Produit (lhs, rhs) -> (eval_expr lhs) * (eval_expr rhs)
17     | Somme (lhs, rhs) -> (eval_expr lhs) + (eval_expr rhs)
18     | Immédiat x -> x
19
20 ;;
21 val eval_expr : expression -> int = <fun>
22 # eval_expr e;;
23 - : int = 114
    
```

Vous ne rêvez pas, en 10 lignes, j'ai déclaré, instancié et évalué un AST.

