**Informatique** *Cours*
**Programmation fonctionnelle avec OCaml - Données structurées** Ipesup

## Objectifs

- Comprendre la manipulation des données structurées en OCaml.
- Savoir utiliser les paires et les $n$-uplets pour regrouper des valeurs.
- Découvrir les enregistrements (produits nommés) et leur intérêt par rapport aux $n$-uplets.
- Maîtriser la définition et l'utilisation des types énumérés (sommes disjointes).
- Pratiquer le filtrage par motifs (`pattern-matching`), y compris avec le motif joker et les motifs multiples.

Une donnée structurée est constituée d'un assemblage de valeurs. Des exemples classiques de telles données sont les dates ou les fiches (contacts téléphoniques, etc.).

On a trois types de données structurées en OCaml :

- les types produits (paires et $n$-uplets),
- les types produits nommés (enregistrements)
- les types sommes (énumérations et sommes disjointes).

# 1 Paires et $n$-uplets

### Paires

Une paire est constituée de l'assemblage de deux expressions `<expr1>` et `<expr2>`. Pour construire une paire, on écrit `(<expr1>, <expr2>)`.

**Exemple**

```
# let p1 = (1, 2) ;;
val p1 : int * int = (1, 2)
```

Le type d'une paire est un produit noté $\tau_1 * \tau_2$, où $\tau_1$ et $\tau_2$ sont les types respectifs des expressions `<expr1>` et `<expr2>`.
Les types des expressions qui constituent une paire peuvent être différents.

**Exemple**

```
# let p2 = ('a', 2.7 +. 2.2) ;;
val p : char * float = ('a', 4.9)
```

Une paire peut également contenir d'autres paires.

**Exemple**

```
# let p3 = (p1, p2) ;;
val p3 : (int * int) * (char * float) = ((1, 2), ('a', 4.9))
```

## Accès aux composantes

Une première façon d'accéder aux composantes d'une paire est d'appeler l'une des deux fonctions prédéfinies `fst` ou `snd`, qui permettent respectivement d'extraire la composante de gauche et la composante de droite d'une paire.

**Exemple**

```
# let x = fst p1 ;;
val x : int = 1

# let y = snd p1 ;;
val y : int = 2
```

La deuxième solution consiste à utiliser `let` avec un motif `(x, y)` comme ci-dessous :

**Exemple**

```
# let (x, y) = p1 ;;
val x : int = 1
val y : int = 2
```

Cette forme de déclaration permet de déconstruire une paire, en donnant un nom à chaque composante. Le motif peut être aussi complexe que n'importe quelle construction de paire.

**Exemple**

```
# let ((x, y), z) = p3 ;;
val x : int = 1
val y : int = 2
val z : char * float = ('a', 4.9)
```

Il est parfois utile de déconstruire une paire pour ne récupérer que quelques composantes, en ignorant les autres. Pour ne pas introduire de noms de variables inutiles, on peut utiliser le symbole `_`.

**Exemple**

```
# let ((_, y), _) = p3 ;;
val y : int = 2
```

### Les $n$-uplets

Pour regrouper un nombre quelconque de valeurs, on utilise des $n$-uplets `(<expr1>, ..., <exprn>)`. Le type d'un $n$-uplet est un produit $\tau_1 * \cdots * \tau_n$ et, comme pour les paires, les types $\tau_i$ sont quelconques.

**Exemple**

```
# let t = ('a', 1.2, (true, 0)) ;;
val t : char * float * (bool * int) = ('a', 1.2, (true, 0))
```

L'accès aux éléments d'un $n$-uplet se fait en utilisant la construction `let` avec motifs.

**Exemple**

```
# let (x, _, (_, y)) = t ;;
val x : char = 'a'
val y : int = 0
```

**Attention**

Pas confondre des types similaires qui représentent des structures différentes :

`int * int * int` `(int * int) * int` `int * (int * int)`

Un $n$-uplet peut être passé en argument à une fonction ou renvoyé comme résultat. Pour simplifier l'accès à ses composantes, les expressions fonctionnelles sont étendues avec la syntaxe : `fun <motif> -> <expr>`

**Exemple**

```
let f (x, y, (a, b)) = x + y * a - b
```

## 2 Enregistrements

La structure de $n$-uplets a deux défauts majeurs.

— Le type produit ne donne pas assez d'informations pour identifier avec précision les objets qu'il représente.
— Un $n$-uplet avec beaucoup de composantes devient très vite compliqué à utiliser en pratique.

**Exemple**

Pour une base de données regroupant des informations sur des personnes : nom, prénom, adresse, date de naissance, téléphone fixe, téléphone portable.

```
let v =
  ("Durand","Jacques",
   ("2 rue J.Monod","Orsay Cedex", 91893),
   (10,03,1967), "0130452637","0645362738")
```

La consultation des composantes sous cette forme est pénible. De plus, il est très facile de confondre les composantes de ces $n$-uplets, qui ont le même type, mais qui représentent des informations bien différentes.

## 2.1 Produits nommés

Pour résoudre ces problèmes, on préfère utiliser des **produits nommés**, aussi appelés **enregistrements** (ou *records* en anglais). Un produit nommé est un $n$-uplet où les composantes ont chacune un identificateur distinct. Par ailleurs, OCaml exige de donner un nom à chaque produit nommé.

**Exemple**

```
type complex = { re : float ; im : float }
```

La définition ci-dessus définit le type `complex` comme un produit nommé avec deux champs `re` et `im`, chacun de type `float`.

Les produits nommés sont de la forme suivante :

```
{ <champ_1> : <type_1> ; ... ; <champ_n> : <type_n> }
```

où les champs `<champ_i>` ont des noms distincts.

De plus, la règle de portée lexicale pour les champs est la même que pour les déclarations, c'est-à-dire que le nom d'un champ peut masquer un champ d'une définition précédente.

Il y a deux manières de créer des produits nommés.

1. Donner les valeurs de chaque champ en utilisant la syntaxe suivante :

   ```
   { <champs_1> = <expr_1> ; ... ; <champs_n> = <expr_n> }
   ```

   Pour que cette expression soit bien typée, il faut que chaque expression `<expr_i>` ait le type `<type_i>` attendu pour le champ `<champ_i>`.
2. Utiliser la notation `with` pour créer un produit nommé à partir d'un autre produit (du même type). La forme générale de cette notation est la suivante :

   ```
   { <expr> with <champs_i> = <expr_i> ; ... ; <champs_k> = <expr_k> }
   ```

L'ordre dans lequel les champs sont donnés n'a pas d'importance.

### Exemple

Les deux déclarations suivantes sont équivalentes :

```
let c3 = { im = 0.5 ; re = 1.2 }
let c2 = { re = 1.2 ; im = 0.5 }
```

Les deux valeurs seront bien considérées comme étant égales.

## Comparaison des produits nommés

La relation d'ordre utilisée pour comparer des enregistrements d'un produit nommé *est* l'ordre lexicographique établi selon l'ordre des champs au moment de la définition de `t`.

### Exemple

Si on definit

```
type t = { b : int ; a : int }
```

et après

```
let v1 = { a = 1 ; b = 4 }
let v2 = { b = 2 ; a = 2 }
let v3 = { b = 3 ; a = 2 }
```

alors on a `v2 < v3 < v1`.Par contre, si on avait defini

```
type t = { a : int ; b : int }
```

alors `v1 < v2 < v3`.

## Accès aux champs

L'accès aux champs d'un produit nommé peut se faire de deux manières :

— par notation pointée `<expr>.<champ>` ;

```
        c1.im
```

— par une déclaration avec filtrage par motif.

```
        type t = { a : int ; b : float * char ; c : string }
        let v = { a = 1 ; b = (3.4,'a') ; c = "bonjour" }

        let { b = (_,x) ; c = y } = v
```

La notation avec filtrage par motif est utilisable dans les définitions de fonctions. Ainsi, on peut écrire

```
let f { b = (x, _) } = x *. 0.4
```

pour filtrer des valeurs de type `t` en arguments d'une fonction. Par ailleurs, ce motif permet à l'algorithme de typage d'OCaml d'inférer automatiquement que `f` a le type `t -> float`.

Pour comparer $n$-uplets et enregistrements, on reprend ci-dessous l'exemple des fiches d'une base de donnée.

**Exemple**

```
type address = { street : string ; city : string ; postal_code : int }

type date = { day : int ; month : int ; year : int }

type form = {
  last_name : string ;
  first_name : string ;
  address : address ;
  birthday : date ;
  phone : string ;
  mobile : string ;
}
```

L'intérêt de cette représentation est qu'elle permet un accès plus simple et sens ambiguïté.

# 3 Énumérations

**Définition**

Les *énumérations* sont utilisées pour représenter des ensembles de valeurs appartenant à un domaine fini.

**Exemple**

Prenons l'exemple d'un jeu de cartes.
Pour représenter les quatre enseignes *Pique*, *Coeur*, *Carreau* ou *Trèfle*, on définit le type `enseigne` :

```
type enseigne = Pique | Coeur | Carreau | Trefle
```

Cette définition peut se lire de la manière suivante : « une valeur de type `enseigne` est soit la valeur `Pique`, soit `Coeur`, soit `Carreau` ou bien `Trefle` ».

D'un point de vue ensembliste, le type `enseigne` correspond à un ensemble (fini) qui est défini comme la somme des valeurs qui sont deux à deux disjointes. C'est la raison pour laquelle ce genre de définition est appelée aussi *somme disjointe*.

Les valeurs du type `enseigne` commencent toutes par une **lettre majuscule** : c'est ce qui permet à OCaml de distinguer une valeur d'un type et des variables (qui commencent toujours par une minuscule).

## 3.1 Constructeurs

Les constantes `Pique`, `Coeur`, `Carreau` et `Trefle`, utilisées pour onstruire des valeurs de type `enseigne`, sont appellées **constructeurs** du type `enseigne`.

L'ordre (de gauche à droite) utilisé pour déclarer les constructeurs induit un ordre sur ces valeurs. C'est cette relation d'ordre qui est utilisée par les opérateurs de comparaison prédéfinis d'OCaml.

Les constructeurs d'une énumération peuvent être utilisés dans n'importe quelles expressions ou structures de données.

**Exemple**

Le type `ens` ci-dessous définit des enregistrements avec un champ `e` qui contient une valeur de type `enseigne`.

```
type ens = { e : enseigne ; m : string }
let v = { e = Pique ; m = "pique" }
```

Lorsqu'on manipule une valeur appartenant à une somme disjointe, on effectue généralement une analyse par cas pour savoir de quel constructeur il s'agit, puis on réalise un branchement.

```
if v.e = Pique then
  <expr_1>
else if v.e = Coeur then
  <expr_2>
else if v.e = Carreau then
  <expr_3>
else
  <expr_4>
```

Plutôt que cette « cascade » de if-then-else, on préfère écrire cette analyse par cas en utilisant l'instruction de filtrage `match-with` :

```
match v.e with
| Pique -> <expr_1>
| Trefle -> <expr_2>
| Coeur -> <expr_3>
| Carreau -> <expr_4>
```

La construction `match-with` permet de regrouper des motifs de filtrage sur une même branche.

**Motif *ou***

Le filtrage par motif ou permette de factoriser des morceaux de code quand un même traitement doit être réalisé pour plusieurs constructeurs.

**Exemple**

On peut regrouper les cas `Pique` et `Trefle` :

```
match v.e with
| Pique | Trefle -> <expr_1>
| Coeur -> <expr_3>
| Carreau -> <expr_4>
```

### Motif joker

Représenté par le symbole _, ce motif représente tous les cas possibles.

#### Exemple

On peut factoriser tous les cas autres que `Pique` et `Trefle` de la manière suivante.

```
match v.e with
| Pique -> <expr_1>
| Trefle -> <expr_2>
| _ -> <expr_joker>
```

L'intérêt principal d'utiliser une construction `match-with` plutôt qu'une cascade de `if-then-else` est que le compilateur OCaml effectue une **analyse d'exhaustivité**, c'est-à-dire qu'il vérifie que tous les cas ont été couverts.

#### Exemple

Par exemple, si on oublie de traiter le constructeur `Coeur` comme ci-dessous :

```
match v.e with
| Pique -> <expr_1>
| Trefle -> <expr_2>
| Carreau -> <expr_4>
```

alors le compilateur émet le message d'avertissement suivant :

```
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Coeur
```

Cela indique que le filtrage n'est pas exhaustif, et qu'il manque le traitement du constructeur `Coeur`.

Le compilateur peut également détecter les cas inutiles dans un filtrage.

#### Exemple

L'expression `<expr_1_bis>` associée à la deuxième branche pour le motif `Pique` ne sera jamais évaluée puisque ce cas est déjà filtré à la première branche.

```
match v.e with
| Pique -> <expr_1>
| Trefle -> <expr_2>
| Coeur -> <expr_3>
| Pique -> <expr_1_bis>
| Carreau -> <expr_4>
```

Le compilateur détecte cette erreur et émet le message suivant à l'utilisateur :

```
File "test.ml", line 25, characters 4-9:
25 | | Pique -> ()
       ^^^^^
Warning 11 [redundant-case]: this match case is unused.
```

## 3.2 Sommes disjointes avec arguments

Les constructeurs d'une somme disjointe peuvent également avoir des arguments.

**Exemple**

Pour représenter les valeurs de cartes roi, reine, valet, et points (ou petites cartes), on définit le type `carte` suivant :

```
type carte = Roi | Reine | Valet | Point of int
```

Ici une carte est soit un `Roi`, soit une `Reine`, soit un `Valet`, soit une carte `Point x`. Le constructeur `Point` est associé à un argument de type `int` : pour construire une carte à l'aide de ce constructeur, il faut l'appliquer à une valeur entière.

```
let c = Point 4
```

### Filtrage

L'accès aux valeurs associées au constructeur `Point` se fait en utilisant `Point x` dans une construction de filtrage, que peut être fait en profondeur.

**Exemple**

```
match c with
| Roi -> <expr_1>
| Reine -> <expr_2>
| Valet -> <expr_3>
| Point 1 -> <expr_as>
| Point x -> <expr_autres_points>
```

Le motif joker peut également être utilisé pour filtrer les arguments des constructeurs.

**Exemple**

Si la valeur des cartes `Point` n'a pas d'importance

```
match c with
| Roi -> <expr_1>
| Reine -> <expr_2>
| Valet -> <expr_3>
| Point 1 -> <expr_as>
| Point _ -> <expr_autres_points>
```

Le compilateur OCaml est capable de détecter des cas non traités dans un filtrage pour des constructeurs avec arguments.

## 3.3 Arguments multiples

Il est parfois nécessaire d'associer plusieurs arguments à un constructeur. OCaml dispose d'une syntaxe pour associer directement plusieurs arguments à un constructeur.

— Les coordonnées des points du plan sont représentées avec un type `coord`. Pour construire des points, on utilise le constructeur `Point` avec un seul argument de type `coord`.

```
type coord = { x : int ; y : int }
type forme = Point of coord
```

— Le constructeur `Cercle` a lui deux arguments, le premier de type `coord` qui contient les coordonnées du centre du cercle, et le second de type `int` qui est le rayon du cercle.

```
type coord = { x : int ; y : int }
type forme = Cercle of coord * int
```

**Exemple**

On peut créer un point de coordonnées (10, 10), ainsi qu'un cercle de centre (50, 100) et de rayon 5 de la manière suivante :

```
let f1 = Point {x = 10; y = 10}
let f2 = Cercle ({x = 50; y = 100}, 5)
```

## Filtrage

Pour accéder aux arguments de ces constructeurs, on utilise la construction de filtrage en profondeur.

**Exemple**

Si `<expr>` est de type `forme`, on peut récupérer toutes les valeurs :

```
match <expr> with
| Point {x = x; y = y} -> <expr_point>
| Cercle ({x = x; y = y}, r) -> <expr_cercle>
```

Les motifs possibles pour filtrer ces valeurs sont les mêmes que ceux utilisés pour les $n$-uplets. Ainsi, on peut spécifier partiellement les noms des champs d'un enregistrement, utiliser le motif joker, etc.

**Exemple**

```
match <expr> with
| Point {x = x} -> <expr_point>
| Cercle (p, _) -> <expr_cercle>
```

le motif `{x = x}` ne récupère que la composante $x$ d'un point, tandis que le motif `(p, _)` récupère uniquement l'enregistrement $p$ associé au centre du cercle.