UNIVERSITÉ Grenoble Alpes

INF201
Algorithmique et Programmation Fonctionnelle
Cours 3: Compléments sur les Types

Année 2022

Définir ses propres types ?

Langages de programmation
$\Rightarrow$ définir de nouveaux types à partir de types existants.

Forme générale d'une définition de type en OCaml :

```
type t = ... (* definition d'un nouveau type *)
```

Le type `t` peut ensuite être utilisé pour définir :

- des identificateurs
- des fonctions
- ou de nouveaux types ...

$\Rightarrow$ on va voir différents mécanismes de définition de types en OCaml ...

Plan

Types synonymes

Types énumérés

Types produit

Type somme (ou union)

Définir un type synonyme

Motivations : **renommer** un type existant

- types spécifiques à un contexte
- plus facile à se rappeler, à réutiliser
- rend le code plus lisible

Définir un type synonyme

Motivations : **renommer** un type existant

- types spécifiques à un contexte
- plus facile à se rappeler, à réutiliser
- rend le code plus lisible

Syntaxe générale :

```
type nouveau_type = type_existant
     (* ajout possible de contraintes ``informatives'' *)
```

Définir un type synonyme

Motivations : **renommer** un type existant

- types spécifiques à un contexte
- plus facile à se rappeler, à réutiliser
- rend le code plus lisible

Syntaxe générale :

```
type nouveau_type = type_existant
     (* ajout possible de contraintes ``informatives'' *)
```

Exemple : Change

- `type euros = float`
- `type dollars = float`
- `type taux = int (* [0,100] *)`
- Fonction qui convertit des euros en dollars
  - Description: `convertit(e,t)` est le prix `e` convertit selon `t`%
  - Profil: `convertit` : `euros` → `taux` → `dollars`

Intérêt : plus lisible qu'une signature "anonyme"

`convertit` : `float` → `int` → `float`

Outline

Types synonymes

Types énumérés

Types produit

Type somme (ou union)

Types énumérés

Exemple : comment exprimer

- l'enseigne d'une carte :$\{\spadesuit, \heartsuit, \diamondsuit, \clubsuit\}$
- la couleur d'une carte :$\{\text{rouge}, \text{noir}\}$
- la hauteur d'une carte :$\{\text{as}, \text{deux}, \text{trois}, \ldots, \text{dame}, \text{roi}\}$

Du point de vue mathématique : ensemble défini en extension
$\hookrightarrow$ par *énumération explicite* de leurs éléments . . .

Types énumérés

Exemple : comment exprimer

- l'enseigne d'une carte :$\{\spadesuit, \heartsuit, \diamondsuit, \clubsuit\}$
- la couleur d'une carte :$\{\text{rouge}, \text{noir}\}$
- la hauteur d'une carte :$\{\text{as}, \text{deux}, \text{trois}, \ldots, \text{dame}, \text{roi}\}$

Du point de vue mathématique : ensemble défini en extension
$\hookrightarrow$ par *énumération explicite* de leurs éléments . . .

Définir un type énuméré en OCaml:

```
type nouveau_type = Valeur_1 | Valeur_2 | ... | Valeur_n
```

Remarque

- les noms des valeurs doivent débuter par une Majuscule
- `nouveau_type` est un *type énuméré*
- `Valeur_1`, ..., `Valeur_n` sont des *constantes symboliques*
- `Valeur_1`, ..., `Valeur_n` sont de type `nouveau_type`
- Ordre implicite entre les constantes (issu de la définition du type)

Types énumérés : Quelques exemples

Jeu de cartes

Exemple : Quelques couleurs

```
type palette = Rouge | Bleu | Jaune
```

Exemple : Types pour un jeu de cartes

```
type enseigne = Trefle | Coeur | Carreau | Pique
type couleur = Rouge | Noir
```

Types énumérés : Quelques exemples

Jeu de cartes

Exemple : Quelques couleurs

```
type palette = Rouge | Bleu | Jaune
```

Exemple : Types pour un jeu de cartes

```
type enseigne = Trefle | Coeur | Carreau | Pique
type couleur = Rouge | Noir
```

Exemple : Couleur associée à une enseigne

$\hookrightarrow$ renvoie la couleur associée à l'enseigne d'une carte

Types énumérés : Quelques exemples

Jeu de cartes

Exemple : Quelques couleurs

```
type palette = Rouge | Bleu | Jaune
```

Exemple : Types pour un jeu de cartes

```
type enseigne = Trefle | Coeur | Carreau | Pique
type couleur = Rouge | Noir
```

Exemple : Couleur associée à une enseigne

↪ renvoie la couleur associée à l'enseigne d'une carte

- Description: `couleurEnseigne` renvoie la couleur d'une enseigne
  - Coeur et Carreau sont de couleur Rouge
  - Pique et Trefle sont de couleur Noir
- Signature: `couleurEnseigne`: `enseigne` → `couleur`
- Exemples: `couleurEnseigne Pique = Noir`, ...

DEMO: Implémentation de couleurEnseigne

Une nouvelle primitive : **filtrage** ou **pattern-matching**

Ton meilleur ami . . .

Une nouvelle primitive : **filtrage** ou **pattern-matching**

Ton meilleur ami . . .

le pattern-matching = évaluation par **analyse de cas**

Une nouvelle primitive : **filtrage** ou **pattern-matching**

Ton meilleur ami . . .

le pattern-matching = évaluation par **analyse de cas**

Syntaxe :

```
match expression with
  | pattern_1 → expression_1
  | pattern_2 → expression_2
    ...
  | pattern_n → expression_n
```

Une nouvelle primitive : **filtrage** ou **pattern-matching**

Ton meilleur ami . . .

le pattern-matching = évaluation par **analyse de cas**

Syntaxe :

```
match expression with
  | pattern_1 → expression_1
  | pattern_2 → expression_2
    ...
  | pattern_n → expression_n
```

Signification :

- `expression` est évaluée puis comparée aux différents `pattern_i`, dans l'ordre (le "matching" dépend du type de `expression`)
- l'`expression_i`, associée au 1er `pattern_i` qui "match" est renvoyée

Une nouvelle primitive : **filtrage** ou **pattern-matching**

Ton meilleur ami . . .

le pattern-matching = évaluation par **analyse de cas**

Syntaxe :

```
match expression with
   | pattern_1 → expression_1
   | pattern_2 → expression_2
     ...
   | pattern_n → expression_n
```

Signification :

- `expression` est évaluée puis comparée aux différents `pattern_i`, dans l'ordre (le "matching" dépend du type de `expression`)
- l'`expression_i`, associée au 1er `pattern_i` qui "match" est renvoyée

Remarque

- La 1ère barre verticale est optionnelle
- on peut utiliser le symbole _ (pour "*autre choix*") à la place de `pattern_n`

(Pattern) Matching sur un exemple

Le jeu de carte

Exemple : `couleurEnseigne` avec un `if...then...else`

```
let couleurEnseigne (e : enseigne) : couleur =
  if (e=Pique || e = Trefle) then Noir
  else (* necessairement e = Coeur || e = Carreau *)
   Rouge
```

(Pattern) Matching sur un exemple

Le jeu de carte

Exemple : `couleurEnseigne` avec un `if...then...else`

```
let couleurEnseigne (e : enseigne) : couleur =
  if (e=Pique || e = Trefle) then Noir
  else (* necessairement e = Coeur || e = Carreau *)
   Rouge
```

Exemple : `couleurEnseigne` par pattern-matching

```
let couleurEnseigne (e : enseigne) : couleur =
  match e with
    | Pique → Noir
    | Trefle → Noir
    | Coeur → Rouge
    | Carreau → Rouge
```

(Pattern) Matching sur un exemple

version plus concise . . .

Exemple : `couleurEnseigne` avec un pattern-matching plus concis

```
let couleurEnseigne (e : enseigne) : couleur =
  match e with
    Pique | Trefle → Noir
    | Coeur | Carreau → Rouge
```

(Pattern) Matching sur un exemple

version plus concise . . .

Exemple : `couleurEnseigne` avec un pattern-matching plus concis

```
let couleurEnseigne (e : enseigne) : couleur =
  match e with
    Pique | Trefle → Noir
    | Coeur | Carreau → Rouge
```

Exemple : `couleurEnseigne`, version encore plus concise

```
let couleurEnseigne (e : enseigne) : couleur =
  match e with
    Pique | Trefle → Noir
    | _ → Rouge
```

Filtrage pour les types énumérés

Pour traiter un élément du type énuméré

```
type nouvtype = Valeur_1 | Valeur_2 | ... | Valeur_n
```

on peut associer le schéma de pattern matching suivant :

```
match expression with (* expression de type nouvtype *)
  | Valeur_1 → expression_1
  | Valeur_2 → expression_2
   ...
  | Valeur_n → expression_n
```

Filtrage pour les types énumérés

Pour traiter un élément du type énuméré

```
type nouvtype = Valeur_1 | Valeur_2 | ... | Valeur_n
```

on peut associer le schéma de pattern matching suivant :

```
match expression with (* expression de type nouvtype *)
  | Valeur_1→ expression_1
  | Valeur_2 → expression_2
   ...
  | Valeur_n → expression_n
```

Règles

- Le filtrage s'appuie sur la définition du type
- Les `expression_i` pour $i \in \{1, \ldots, n\}$ doivent être de même type
- Le pattern-matching doit être exhaustif (ou utiliser le symbole _)

```
match expression with
  | Valeur_1→ expression_1
   ...
  | _ → expression
```

Entrainement sur les types énumérés

Exercice

- Définir un type énuméré `mois` qui décrit les 12 mois de l'année
- Définir la fonction `nbre_de_jour`: `mois` $\rightarrow$ `int` qui associe à chaque mois son nombre de jours

Pattern-matching sur des types prédéfinis

Le pattern-matching est une généralisation du `if...then...else...`
↪ s'applique aussi à des types prédéfinis : int, bool, float

Pattern-matching sur des types prédéfinis

Le pattern-matching est une généralisation du `if...then...else...`
↪ s'applique aussi à des types prédéfinis : int, bool, float

Exemple : Est-ce qu'un entier est un nombre premier inférieur à 20 ?

```
let est_premier_inf_20 (n : int) : bool =
 match n with
  | 1 | 3 | 5 | 7 | 11 | 13 | 17 | 19 → true
  | _ → false
```

Pattern-matching sur des types prédéfinis

Le pattern-matching est une généralisation du `if...then...else...`
↪ s'applique aussi à des types prédéfinis : int, bool, float

Exemple : Est-ce qu'un entier est un nombre premier inférieur à 20 ?

```
let est_premier_inf_20 (n : int) : bool =
 match n with
  | 1 | 3 | 5 | 7 | 11 | 13 | 17 | 19 → true
  | _ → false
```

Exemple : Est-ce qu'un caractère est une voyelle majuscule

```
let est_voyelle_majuscule (c:char) : bool = match c with
  |'A' | 'E' | 'I' | 'O' | 'U' | 'Y' → true
  | _ → false
```

Pattern-matching sur des types prédéfinis

Le pattern-matching est une généralisation du `if...then...else...`
↪ s'applique aussi à des types prédéfinis : int, bool, float

Exemple : Est-ce qu'un entier est un nombre premier inférieur à 20 ?

```
let est_premier_inf_20 (n : int) : bool =
 match n with
  | 1 | 3 | 5 | 7 | 11 | 13 | 17 | 19 → true
  | _ → false
```

Exemple : Est-ce qu'un caractère est une voyelle majuscule

```
let est_voyelle_majuscule (c:char) : bool = match c with
  |'A' | 'E' | 'I' | 'O' | 'U' | 'Y' → true
  | _ → false
```

Exemple : Le pattern-matching sur des réels est à éviter . . .

```
match 4.3 −. 1.2 with
  3.1 → true
  _   → false
```

⇝ renvoie `false`

Un raccourci dans l'écriture de pattern-matching

cas des caractères

"Profiter de l'ordre entre les caractères"

```
match expression
    with
 ...
 | p1 → v
 | p2 → v
 ...
 | pm → v
 ....
```

$\rightsquigarrow$

```
match expression
    with
 ...
 | p1 .. pm → v
 ....
```

où `p1`, .., `pm` sont des caractères *consécutifs* et `p1` et `pm` en sont les valeurs minimales et maximales (dans n'importe quel ordre)

Un raccourci dans l'écriture de pattern-matching

cas des caractères

"Profiter de l'ordre entre les caractères"

```
match expression
    with
 ...
 | p1 → v
 | p2 → v
  ...
 | pm → v
  ....
```

$\rightsquigarrow$

```
match expression
    with
 ...
 | p1 .. pm → v
 ....
```

où `p1`, .., `pm` sont des caractères *consécutifs* et `p1` et `pm` en sont les valeurs minimales et maximales (dans n'importe quel ordre)

Exemple : "Profiter de l'ordre entre les caractères"

```
let est_majuscule (c:char) : bool = match c with
   'A' .. 'Z' → true
   | _ → false
```

Outline

Types synonymes

Types énumérés

Types produit

Type somme (ou union)

Types produit : exemple(s)

Exemple : Nombres complexes

Comment spécifier des nombres complexes ?

En maths, on définit :

$$\mathbb{C} \quad = \{a + ib \mid a \in \mathbb{R}, b \in \mathbb{R}\}$$

Types produit : exemple(s)

Exemple : Nombres complexes

Comment spécifier des nombres complexes ?

En maths, on définit :

$$\mathbb{C} \quad = \{a + ib \mid a \in \mathbb{R}, b \in \mathbb{R}\}$$

z	a	b
$3.0 + i * 2.5$	3.0	2.5
$12.0 + i * 1.5$	12.0	1.5
$(1.0 + i) * (1.0 - i)$		

Types produit : exemple(s)

Exemple : Nombres complexes

Comment spécifier des nombres complexes ?

En maths, on définit :

$$\mathbb{C} \quad = \{a + ib \mid a \in \mathbb{R}, b \in \mathbb{R}\}$$

z	a	b
$3.0 + i * 2.5$	3.0	2.5
$12.0 + i * 1.5$	12.0	1.5
$(1.0 + i) * (1.0 - i)$		

En fait, on peut aussi définir :

$$\boxed{\mathbb{C} = \mathbb{R} \times \mathbb{R}}$$

L'opération $\times$ est le produit cartésien d'ensembles

Exemple : Définir des cartes

Définition d'un type carte comme un produit cartésien ?

Produit (cartésien) de types

Un produit cartésien de types permet de représenter des couples d'éléments de type différents

Type	Valeur
$\alpha * \beta$	(•,•)
`int*int`	(1,2)
`int*float`	(1,2.0)

Produit (cartésien) de types

Un produit cartésien de types permet de représenter des couples d'éléments de type différents

Type	Valeur
$\alpha * \beta$	(•,•)
`int*int`	(1,2)
`int*float`	(1,2.0)

Définir de nouveaux produits de type :

```
type nouveau_type = type_existant1 * type_existant2
```

Produit (cartésien) de types

Un produit cartésien de types permet de représenter des couples d'éléments de type différents

Type	Valeur
$\alpha * \beta$	(•,•)
`int*int`	(1,2)
`int*float`	(1,2.0)

Définir de nouveaux produits de type :

```
type nouveau_type = type_existant1 * type_existant2
```

Deux opérateurs prédéfinis sur les couples :

- `fst(`$\bullet_1$`,`$\bullet_2$`)` = $\bullet_1$
- `snd(`$\bullet_1$`,`$\bullet_2$`)` = $\bullet_2$

Nommer les composants d'un produit cartésien :

```
let (x1,x2) = v in expression_utilisant_x1_et_x2
```

$\hookrightarrow$ definit les identificateurs locaux `x1` et `x2` à partir de la valeur `v` de type produit cartésien

DEMO: Produit de types

Généralisation du produit cartésien de types

Même principe

Peut être généralisé à des *n*-uplets :

- définition du type : `let mon_type = type1 * type2 * ... * typen`
- *construction* de valeurs : `(v1,v2,...,vn)`
- *deconstruction* de valeurs (nommer les composants) :

  `let (x1,...,xn) = v in expression`

  où :
  - `expression` utilise `x1,...,xn`
  - `v` est une valeur de type produit cartésien à n composants

DEMO: Produit de types généralisés

Entrainement sur les produits de type

Exercice : Se familiariser avec les n-uplets

- Définir le type `couple_entiers` qui définit un couple d'entiers
- Définir la fonction `echange` qui échange les entiers d'un `couple_entiers`
- Définir la fonction `my_fst` qui se comporte comme la fonction prédéfinie `fst` sur le type `couple_entiers`

Exercice sur les nombres complexes

- Définir un type `complexe` correspondant aux nombres complexes
- Définir la fonction `partie_reelle` de type `complexe` $\rightarrow$ `float` qui renvoie la partie réelle d'un complexe
- Même question pour la partie imaginaire
- Définir la fonction `conjugue` : `complexe` $\rightarrow$ `complexe`
  Rappel : le conjugué de $a + b.i$ est $a - b.i$

Entrainement sur les produits de type (suite)

Un peu de géométrie . . .

Exercice sur les vecteurs

- Définir le type `vect` correspondant aux vecteurs du plan
- Définir la fonction `somme` : `vect` $\rightarrow$ `vect` $\rightarrow$ `vect` qui effectue la somme de deux vecteurs
- Quel est le type de la fonction `produit_scalaire` ?
  Définissez cette fonction
  Rappel : le produit scalaire de 2 vecteurs $\overrightarrow{u}$, $\overrightarrow{v}$: $||\overrightarrow{u}||.||\overrightarrow{v}||.\cos(\overrightarrow{u},\overrightarrow{v})$
  avec $\cos(\overrightarrow{u},\overrightarrow{v}) = \frac{u_x.v_x+u_y.v_y}{||\overrightarrow{u}||.||\overrightarrow{v}||}$

Entrainement sur les produits de type (suite)

Un peu de géométrie . . .

Exercice sur les vecteurs

- Définir le type `vect` correspondant aux vecteurs du plan
- Définir la fonction `somme` : `vect` $\rightarrow$ `vect` $\rightarrow$ `vect` qui effectue la somme de deux vecteurs
- Quel est le type de la fonction `produit_scalaire` ?
  Définissez cette fonction
  Rappel : le produit scalaire de 2 vecteurs $\overrightarrow{u}$, $\overrightarrow{v}$: $||\overrightarrow{u}||.||\overrightarrow{v}||.\cos(\overrightarrow{u},\overrightarrow{v})$
  avec $\cos(\overrightarrow{u},\overrightarrow{v}) = \frac{u_x.v_x+u_y.v_y}{||\overrightarrow{u}||.||\overrightarrow{v}||}$
- La rotation par un angle θ d'un vecteur de coordonnées (x, y) est exprimée par la formule :

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

  Définissez la fonction `rotation`: `float` $\rightarrow$ `vect` $\rightarrow$ `vect` telle que `rotation angle v` renvoie l'image du vecteur v par rotation de `angle`

Résumons

avant de continuer . . .

- Ecrire un programme $\Rightarrow$ spécifier/traiter des données
- Langages de programmation : notion de type
  - type = un ensemble de valeurs, des opérations
  - types prédéfinis (`int`, etc.)
  - des mécanismes de construction de type
- On a vu trois formes de construction de type :
  - types synonymes

    ```
    type vitesse = float
    ```

  - types énumérés

    ```
    type couleur = Bleu | Jaune | Rouge
    ```

  - types produit

    ```
    type point = float * float
    ```

Plan

Types synonymes

Types énumérés

Types produit

Type somme (ou union)

Pourquoi encore un nouvelle construction type ?

Mélanger des choux et des carottes . . .

. . . dans le contexte du système de type OCaml

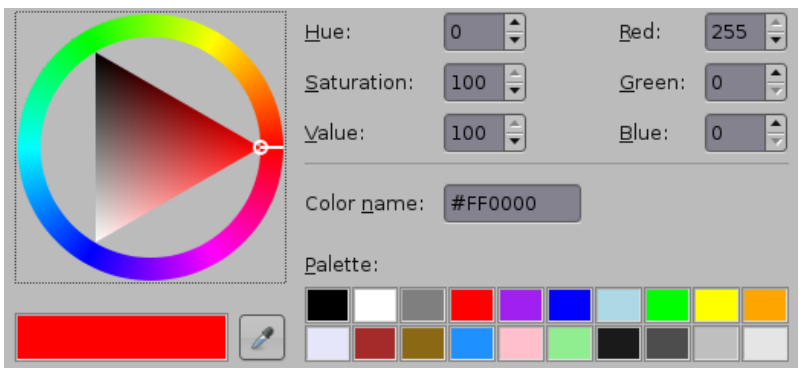
Pourquoi encore un nouvelle construction type ?

Mélanger des choux et des carottes . . .

. . . dans le contexte du système de type OCaml

Quelques concepts que l'on ne sait pas modéliser facilement avec les types actuels :

- Comment définir un type `figure` pour représenter des cercles, des triangles, des quadrilatères ?
- Comment définir un type représentant une palette complète de couleurs ?



- Comment définir un type jeu de carte permettant de jouer à plusieurs jeux ?

Retour sur les couleurs

Présentation du type “Union” par un exemple

Définition de type	Filtrage

`type couleur =`
`  Bleu`
`  | Jaune`
`  | Rouge`
	`let est_bleu (c : couleur) : bool =`
	`  match c with`
	`  | Bleu → true`
	`  | Jaune → false`
	`  | Rouge → false`

Remarque Le type `couleur` contient trois constructeurs “constants” □

Retour sur les couleurs

Présentation du type “Union” par un exemple

Définition de type	Filtrage

`type couleur =`
`  Bleu`
`  | Jaune`
`  | Rouge`
	`let est_bleu (c : couleur) : bool =`
	`  match c with`
	`  | Bleu → true`
	`  | Jaune → false`
	`  | Rouge → false`

Remarque Le type `couleur` contient trois constructeurs “constants” □

→ *Comment ajouter à ce type de nouvelles couleurs qui n’ont pas de* **nom** *mais seulement un* **numéro de référence** *?*

Ex : jaune citron ⇝ 234

Retour sur les couleurs

Présentation du type “Union” par un exemple

Définition de type

```
type couleur =
  | Bleu
  | Jaune
  | Rouge
  | Numero of int
```

Filtrage

```
let est_bleu (c : couleur) : bool =
  match c with
    | Bleu → true
    | Jaune → false
    | Rouge → false
(* Numero dans [100,200] =>
    bleu *)
    | Numero i → i>=100 && i<=200
```

- Le type `couleur` a 3 constructeurs “constants” et un “non constant”.
- `Numero` 234 représente la couleur numérotée 234 (dans 1 catalogue donné)

Retour sur les couleurs

Présentation du type “Union” par un exemple

Définition de type

```
type couleur =
  | Bleu
  | Jaune
  | Rouge
  | Numero of int
```

Filtrage

```
let est_bleu (c : couleur) : bool =
  match c with
    | Bleu → true
    | Jaune → false
    | Rouge → false
(* Numero dans [100,200] =>
    bleu *)
    | Numero i → i>=100 && i<=200
```

- Le type `couleur` a 3 constructeurs “constants” et un “non constant”.
- `Numero` 234 représente la couleur numérotée 234 (dans 1 catalogue donné)

→ *Comment ajouter de nouvelles couleurs définies par leur* **code RVB** *?*
Ex : 30% Rouge, 50% Vert, 20% Bleu

Retour sur les couleurs

Présentation du type “Union” par un exemple

Définition de type	Filtrage

`type couleur =`
`  | Bleu`
`  | Jaune`
`  | Rouge`
`  | Numero of int`
`(* palette RGB *)`
`  | RVB of int * int * int`
	`let est_bleu (c : couleur) : bool =`
	`  match c with`
	`  | Bleu → true`
	`  | Jaune → false`
	`  | Rouge → false`
	`(* Numero dans [100,200] => bleu *)`
	`  | Numero i → i>=100 && i<=200`
	`(* b doit "dominer" *)`
	`  | RVB (r,v,b) → b>(3*v) && b>(3*r)`

- Le type `couleur` a trois constructeurs “constantes” et deux “non-constantes”
- `RVB`(0,0,255) correspond à Bleu
- `RVB`(255,255,0) correspond à Jaune

Types Union : généralisation des types énumérés

Forme générale

Syntaxe d'un type union :

```
type nouveau_type =
  | Identificateur_1 of type_1
  | Identificateur_2 of type_2
  ...
  | Identificateur_n of type_n
```

Types Union : généralisation des types énumérés

Forme générale

Syntaxe d'un type union :

```
type nouveau_type =
  | Identificateur_1 of type_1
  | Identificateur_2 of type_2
  ...
  | Identificateur_n of type_n
```

Remarques :

- `Identificateur_i` est appelé un constructeur de `nouveau_type`
- la définition "`of type_i`" est optionnelle
- `type_i`, peut être n'importe quel type déjà défini
- `Identificateur_i` doit commencer par une lettre majuscule

Types Union : généralisation des types énumérés

Forme générale

Syntaxe d'un type union :

```
type nouveau_type =
  | Identificateur_1 of type_1
  | Identificateur_2 of type_2
  ...
  | Identificateur_n of type_n
```

Remarques :

- `Identificateur_i` est appelé un constructeur de `nouveau_type`
- la définition "`of type_i`" est optionnelle
- `type_i`, peut être n'importe quel type déjà défini
- `Identificateur_i` doit commencer par une lettre majuscule

Déclaration d'une expression de type union `t` :

```
let expression = Identificateur v
```

où

- `Identificateur of t` est un constructeur du type `t`
- `v` est une valeur de type `t`

Exemple de type union : des figures géométriques

Définition du type	Filtrage

`type pt = float * float`
 	
`type figure =`
`  | Rectangle of pt * pt`
`  | Cercle of pt * float`
`  | Triangle of pt * pt * pt`
	`let perimetre (f : figure) : float =`
	`  match f with`
	`  | Rectangle (p1, p2) → ...`
	`  | Cercle (_, r) → ...`
	`  | Triangle (p1, p2, p3) → ...`

```
let p1 = (1.0, 2.0) and p2 = (3.9, 2.7) in Rectangle (p1,p2)
let p1 = (1.3, 2.9) in Cercle (p1,3.6)
```

Exercice

- Definir la fonction `distance`: `pt` → `pt` → `float`
- L'aire d'un triangle de cotés a, b, c est calculable en appliquant la formule suivante :

$$A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)} \quad \text{avec} \quad s = \frac{1}{2} \cdot (a+b+c)$$

  Définissez la fonction `aire`: `figure` → `float`
- Définissez un type `vecteur`, un type `droite`, etc.

Exercice : jouons au Tarot . . .

Les cartes du tarot

- les **figures** : roi, dame, cavalier, valet
- les **basses** : de *un* (as) à *dix*
- les **atouts** : numérotés de *un* à *vingts-et-un*
- l'excuse

De plus :

- les figures et les basses sont de 4 couleurs possibles (coeur, carreau, pique et trêfle)
- les **bouts** sont l'excuse et les atouts *un* (petit) et *vingts-et-un*

Valeur des cartes

- bout ou roi : 5 points
- dame : 4 points ; cavalier : 3 points ; valet : 2 points
- autres atouts et carte basse : 1 point

→ Définir en CAML un type `carte` représentant les cartes d'un jeu de tarot
→ Ecrire une fonction `valeur : carte -> int`

Différence entre *constructeurs* et *fonctions unaires*

Les constructeurs et les fonctions unaires prennent en paramètre une valeur d'un certain type et renvoient une valeur d'un autre type.

Une *fonction* :

- effectue un calcul
- ne peut être utilisée dans un pattern-matching :
  $\rightarrow$ la valeur de toute fonction est <`fun`>

Un *constructeur de type* :

- construit une valeur
- peut être utilisé pour du pattern-matching

Conclusion

Résumé :

- Des types plus riches, plus expressifs . . .

Type	**Pourquoi ?**
types synonymes	noms de type significatifs
types énumérés	ensembles finis de constantes
types produits	produits cartésiens
types unions	généralisation . . .

- Utiliser du filtrage et du pattern-matching pour définir des fonctions plus complexes (pour chacun de ces types)

Exercice

Trouvez un exemple de donnée qui se représente naturellement comme un type union. Ecrivez une fonction portant sur ce type.