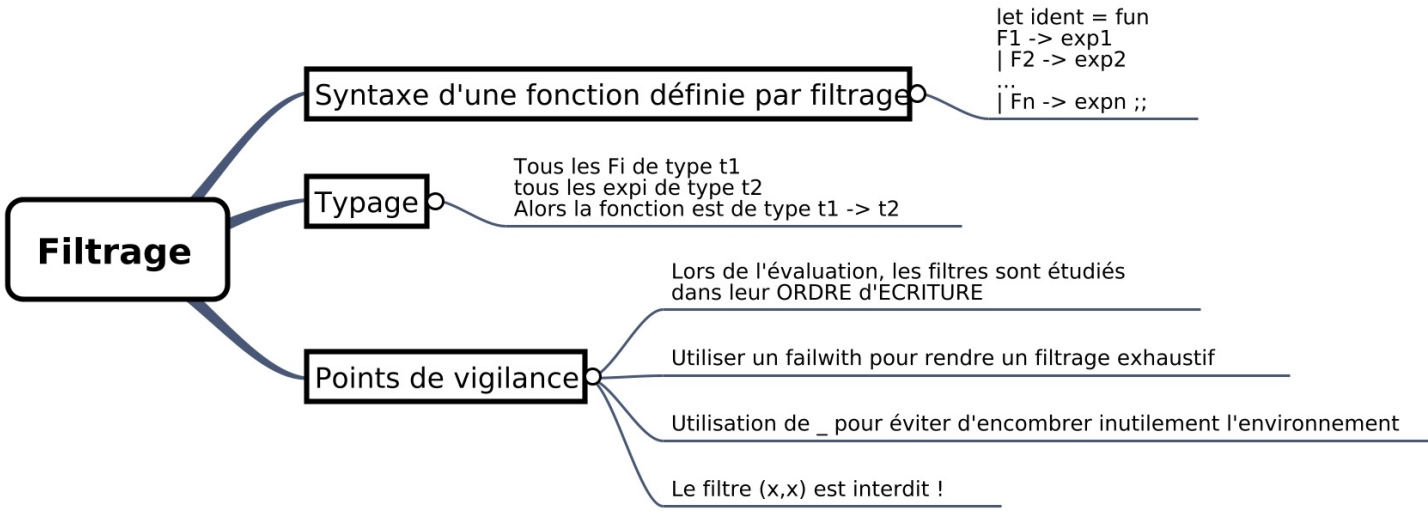
Chapitre 4 : Filtrage

En bref



1. Filtre, généralités

Un filtre permet de préciser les différentes valeurs que peut prendre un paramètre.

Exemple 1

```
let f = function
   0. -> 1.0
| x -> sin(x) /. x  ; ;
(* f  : float -> float = <fun> *)
```

1.1. Filtre, définition

En Caml, un filtre est **une expression constituée de constantes, d'identificateurs, du symbole souligné (_) et des constructeurs de n-uplets**. Les filtres remplace nt et généralisent les paramètres formels de fonctions.

- Les constantes ne laissent passer qu'elles-mêmes
- Les identificateurs et le symbole souligné (_) laissent tout passer (voir plus loin ce qui les distingue)
- les constructeurs de n-uplets permettent de définir la forme l'objet à filtrer

Exemple 2. Exemples de filtre

- 1 est un filtre qui ne laisse passer que la constante 1
- n est un filtre qui laisse passer n'importe quelle valeur de n'importe quel type
- $(0,x)$ et $(0,)$ sont des filtres qui laissent passer tous les couples dont la première composante est nulle.

1.2. Syntaxe d'une fonction définie par filtrage

En Caml, nous utilisons un filtrage pour affiner la définition des fonctions. La syntaxe est la suivante :

```
let ident = function
   F1 -> exp1
| F2 -> exp2
     ...
| Fn -> expn  ; ;
```

Important

Les filtres F_i sont des expressions d'un **même type** t_1, correspondant à différentes valeurs du paramètre formel.
Les expressions exp_i sont différents corps de la fonction, correspondant aux valeurs à prendre en fonction du filtre.

Attention ! Dans un filtre, le nom du paramètre formel n'a aucune influence, mais à l'intérieur d'un même filtre, **on ne peut utiliser qu'une fois un même paramètre formel**. Par exemple, le filtre $(x,0,x)$ est interdit.

Exemple 3

```
let f= function
1->0
| n-> n+1 ; ;
(*f  : int -> int *)
```

```
let zero= function
0->true
| _-> false ; ;
(*f  : int -> int *)
```

```
let divise= function
(_,0) -> false
|(n,d) -> n mod d =0 ; ;
(*f  : int -> bool *)
```

2. Règles d'évaluation

2.1. Typage d'une fonction définie par filtrage

Si **tous** les filtres sont de type t_1 et que **tous** les corps de la fonction sont de type t_2 alors la fonction définie est de type $t_1 \to t_2$. Dans tous les autres cas, il y a une erreur de type.

Remarque : L'évaluation d'une fonction définie par filtrage se fait comme une fonction, mais pour des raisons évidentes, on n'écrira pas la fermeture d'une telle fonction !

2.2. Evaluation d'un appel de fonction définie par filtrage

Lors d'un appel :

```
ident(arg)  ; ;
```

1. L'argument `arg` est évalué dans l'environnement d'appel, on cherche alors à filtrer sa valeur v en parcourant les différents filtres **dans leur ordre d'écriture**.
2. (a) Si aucun filtre ne correspond à l'argument, l'évaluation de la fonction échoue. On dit dans ce cas que le filtrage n'est pas exhaustif.
   Remarquez que le système Caml détecte ces cas lors de la définition de la fonction et affiche une mise en garde :

   Exemple 4

   ```
   let f= function 1->0  |2->3 ; ;
   (*Warning  : this pattern-matching is not exhaustive.
   Here is an example of a case that is not matched  : 0*)

   f(1) ; ;
   (*- :int=0*)

   f(3) ; ;
   (*Exception  : "Match_failure //toplevel// :1 :7"*)
   ```

   Remarque : Ce n'est pas une erreur d'écrire une fonction non exhaustive, mais c'en est une que de l'utiliser en dehors de son ensemble de définition.

   (b) Sinon le filtrage prend fin sur le **premier** filtre (dans l'ordre d'écriture !) correspondant à l'argument :

   ```
   | fi -> expi
   ```

   Là encore, il faut distinguer deux cas :

– **Si `fi` contient des paramètres formels :**

i. Il y a création d'un environnement temporaire contenant des liaisons entre les paramètres formels de `fi` et la valeur correspondante dans l'argument, ajouté à l'environnement de définition de la fonction.

ii. `expi` est alors évalué dans cet environnement et c'est le résultat de la fonction

iii. L'environnement temporaire est détruit

Exemple 5

```
let f = function
   0 -> 1
  |n -> 2*n -3  ; ;

 f(4) ; ;
(*- : int = 5 *)
```

4 n'est pas filtré par 0

4 est filtré par n

1/ Création de `EnvT=[(n,4)><Env-def]`

2/ évaluation de $2 * n - 3$ dans EnvT : 5

3/ destruction de EnvT

Attention ! L'ordre d'écriture des filtres est dont très important ! D'ailleurs, Caml y prend garde :

```
let f=function
  n -> 2*n-3
 |0 -> 1 ; ;
(* Attention  : ce cas de filtrage est inutile. *)

f(0) ; ;
(*- : int = -3 *)
```

Observons un nouvel exemple :

Exemple 6

```
let f=function
   1->"un"
   |n->"plusieurs" ; ;

f(4)  ; ;
```

L'appel à `f(4)` conduit à la création de l'environnement temporaire `EnvT = [(n,4) >< Env-def]` pour y évaluer l'expression constante "plusieurs". La liaison est donc complètement inutile et on cherchera à l'éviter : d'où l'utilisation de _.

– **Si l'expression n'utilise pas le paramètre formel,** il est inutile de créeer la liaison correspondante. C'est dans ce but que l'on filtrera par le caractère souligné

(_) plutôt que par un identificateur. En effet, ce symbole filtre toutes les valeurs, mais ne provoque aucune liaison dans l'environnement temporaire.

2.3. Dernières remarques et erreurs à éviter

Remarque - Si l'on veut rendre exhaustif un filtrage qui ne l'est pas, et personnaliser le message d'erreur à afficher lorsqu'on appelle la fonction en dehors de son ensemble de définition, on peut utiliser la commande

`failwith`

Exemple 7

```
let f = function
   1->"un"
  |2->"deux"
  |_ -> failwith "f n'est définie qu'en 1 et 2."
```

Observez l'usage du symbole souligné pour filtrer tous les cas non traités par les filtres précédents.

Attention ! Un filtre peut distinguer des valeurs particulières (comme 1,2 dans l'exemple précédent), mais il ne permet pas de distinguer des conditions telles que : si l'argument est positif. On sera alors amener à mélanger filtres et sélection.

Exemple 8

```
let repere_plan = function
   (0,0)-> "origine"
   | (x,y)-> if x = y then "première diagonale" else "ailleurs dans le plan"  ; ;
```

Exercice . Exercice TD A

Considérons la fonction

```
let f= function
   (0,_) -> 0
  |(_,0) -> 5
  |(1,x) -> x + 50
  |(x,1) -> x + 35
  | _ -> 3 ; ;
```

Expliquer les calculs de $f(0,1)$, $f(1,0)$, $f(1,4)$, $f(4,1)$, $f(25,5)$.

Exercice . Exercice TP 1

En utilisant des filtrages, réécrire les fonctions booléennes *et*, *ou* et *xor* de type `bool * bool -> bool`.

Exercice . Exercice TP 2

Écrire une fonction `entier : int -> string`, définie par un filtrage exhaustif, qui à un entier n associe la chaîne de caractère "zero" si n vaut 0, "un" si n vaut 1 et dans les autres cas, "pair" si n est pair, et "impair" si n est impair.

```
entier(0) ; ;
-  : string = "zero"
entier(3) ; ;
-  : string = "impair"
```

Exercice . Exercice TP 3

Nous souhaitons distinguer dans le plan réel : l'origine $(0,0)$, les points de l'axe des abscisses, les points de l'axe des ordonnées, et les autres que nous partageons entre les points du demi-plan $x > 0$ et les points du demi-plan $x < 0$.
Écrire une fonction `point : float * float -> string`, obligatoirement définie par filtrage exhaustif, qui à un couple de réels représentant un point du plan associe sa catégorie.

```
point(0.,0.) ; ;
-  : string = "Origine"

point(0.,3.) ; ;
-  : string = "Axe des ordonnées"

point(2.,-3.) ; ;
-  : string = "point du demi plan x>0"
```

Exercice . Exercice TP 4

Écrire une fonction `operation : int * int * char -> int`, obligatoirement définie par un filtrage exhaustif, qui à deux entiers x et y et à un caractère c associe :

- Si c est un des caractères $+, -, *$ ou / : le résultat de l'opération correspondante entre x et y (On veillera à ne pas faire de division par 0)
- Un message d'erreur sinon.

```
operation(7,3,'+') ; ;
-  : int = 10

operation(7,3,'-') ; ;
-  : int = 4

operation(7,0,'/') ; ;
Exception : (Failure division par zéro !)

operation(7,0,' !') ; ;
Exception : (Failure Ce n'est pas une opération connue.)
```

Exercice . Exercice TP 5

La TVA (taxe sur la valeur ajoutée) est une taxe qui s'ajoute à la valeur des marchandises. Elle s'exprime comme un pourcentage du prix de base (appelé prix hors taxe), pourcentage qui dépend de la nature des marchandises et qui est appelé taux de TVA . Les prix TTC (toute taxe comprise) est le prix hors taxe + le montant de la TVA.
Par exemple un produit à 200€(hors taxe) assujetti à une TVA à 12% aura une TVA de 24€et un prix TTC de 224€.

1. Écrire une fonction `prixTTC : float * float -> float` calculant le prix TTC d'une marchandise en fonction de son prix hors taxe et du taux de la TVA.

```
#prixTTC(200.,12.) ; ;
-  : float = 224.0
```

2. Écrire une fonction `prix : string -> float * float`, définie par filtrage, qui à un article de la liste suivante associe son prix unitaire hors taxe et le montant de la TVA :

Article	**Prix unitaire**	**Taux TVA**
pain	1,05	5,5 %
conserve	3,50	7 %
disque	12,30	18,6 %
bijou	356	33 %

   Pour tout autre article, on affichera le message d'erreur "Article indisponible".

3. Écrire une fonction `sommeAPayer : string * int -> float`, utilisant les fonctions précédentes, qui à un nom d'article et au nombre d'articles achetés associe la somme TTC à payer, arrondie à la dizaine de centimes la plus proche (utilisez la fonction `arrondi` du TP 2).

Exercice . Exercice TP 6

Le but de ce problème est d'écrire une fonction qui à une date donnée sous la forme de trois entiers (jour, mois, année) associe le jour de la semaine correspondant à cette date.

1. Écrire une fonction `formule : int * int * int * int -> int` qui à un quadruplet d'entiers (j,m,p,s) associe l'entier

$$j+(48m-1)/5+p/4+p+s/4-2*s$$

2. Écrire une fonction `decoupe : int -> int * int`, qui à un entier n associe son quotient et son reste dans la division entière par 100.

```
decoupe(2014) ; ;
-  : int * int = 20, 14
```

3. Écrire une fonction `deuxMoisAvant : int * int -> int * int`, définie par filtrage, qui étant donné un couple (mois, année) associe le couple correspondant à la

date antérieure de 2 mois à la date donnée.

```
deuxMoisAvant(2,2014) ; ;
-   : int * int = 12, 2013
```

4. Écrire une fonction `leJour : int -> string`, définie par filtrage non exhaustif qui à 0 associe "Dimanche", 1 associe "Lundi"... et à 6 associe "Samedi".

```
leJour(2) ; ;
-   : string = "Mardi"
```

5. Écrire une fonction `modulo7 : int -> int` qui étant donné un entier donné un entier n renvoie le reste de la division euclidienne de n par 7.
Attention ! En Caml, $12 \ mod \ 7$ donne 5 mais $-12 \ mod \ 7$ donne -5. Donc `modulo7(n)` est égal à $n mod 7$ s'il est positif ou nul et $(n \ mod \ 7)+7$ sinon.

```
modulo7(20) ; ;
-   : int = 6
modulo7(-4) ; ;
-   : int = 3
```

6. Il est possible de retrouver le jour de la semaine à partir d'une date donnée par la méthode suivante :
   - Partant de la date (jour, mois, année), on calcule (M,A), le mois et l'année précédent de deux mois la date (mois, année).
     Par exemple, si la date est le $(14,10,2014)$, on obtient $(M,A) = (8,2014)$.
   - On sépare les 4 chiffres de l'année A en un couple (S,P). S correspond aux deux premiers chiffres de A et P à ses deux derniers chiffres.
     Sur notre exemple, on obtient $(S,P) = (20,14)$.
   - On calcule alors un entier K par la formule :

$$K = jour + (48 * M - 1)/5 + P/4 + P + S/4 - 2 * S$$

     On reconnaît la fonction `formule` et on obtient sur notre exemple $K = 72$.
   - Il reste à faire la division euclidienne de K par 7 et à prendre le jour correspondant de la semaine.
     On obtient ici $72 = 7 * 10 + 2$ et le 2ème jour est Mardi.

   Écrire, en utilisant les fonctions précédentes, une fonction `quelJour : int * int * int -> string` mettant en œuvre cette méthode pour calculer le jour correspondant à une date donnée.

```
quelJour(14,7,1789) ; ;
-   : string = "Mardi"
```