Chapitre 4 : Filtrage de motifs

Le filtrage en CAML est une construction à la fois élégante et très puissante. Bientôt, vous ne pourrez plus vous en passer et il faudra alors faire attention à ne pas en abuser.

✔

Le filtrage par motifs permet :

- de gérer différents cas en fonction des valeurs d'une expression ;
- d'accéder aux éléments d'un type construit (tuple, type somme, liste CAML, etc.)
- les deux à la fois !

1 La construction `match ... with`

Définition 1.1

Le filtrage est une *expression* dont la syntaxe est :

OCAML
```
match expr0 with
| motif1 -> expr1
| motif2 -> expr2
...
| motifn -> exprn
```

Toutes les expressions `expr1`, `expr2`, ..., `exprn` doivent être de même type, qui est alors le type de cette expression.

- Comparaison de la *valeur* de `expr0` avec les différents *motifs* ;
- Filtrage effectué *dans l'ordre* et on prend *uniquement* le premier motif qui correspond[1] ;
- Le filtrage doit être *exhaustif* (un des motifs doit correspondre).

✔

Un *motif* est un cas de base d'un type (e.g. l'entier `0`, le caractère `'a'`, la chaîne de caractères `"Pécheux"`) ou une *construction* de CAML (e.g. un tuple, un type somme, etc.)

⚠

Un identifiant (nom de variable) *n'est pas* un motif. Lorsqu'il y a un identifiant dans un motif, il y a une liaison locale.

OCAML
```
let rec factorielle n =
  match n with
  | 0 -> 1
  | m -> m * factorielle (m - 1) (* liaison locale *)
;;
```

OCAML
```
let rec factorielle n =
  match n with
  | 0 -> 1
  | m -> n * factorielle (n - 1) (* liaison local inutile
   ↪  *)
;;
```

1. En anglais on dit qui « match ».

OCaml
```
let rec factorielle n =
  match n with
  | 0 -> 1
  | n -> n * factorielle (n - 1) (* liaison locale !!! *)
;;
```

OCaml
```
let rec factorielle n =
  match n with
  | 0 -> 1
  | _ -> n * factorielle (n - 1) (* _ variable "poubelle"
   ↪  *)
;;
```

On va préférer cette dernière version.

☞ **Remarque 1**

- *| x -> est un « motif » qui correspond toujours !*
- *Liaison locale qui masque une liaison précédente !*
- *On utilise alors la variable spéciale _ : « dans tous les autres cas ».*

OCaml
```
let commence_par_a mot =
  match (String.get mot 0) with
  | 'a' -> true
  | 'A' -> true
  | _ -> false
;;
commence_par_a : string -> bool = <fun>
```

OCaml
```
let commande panier =
  match panier with
  | (_, 0, _) -> "Panier vide."
  | (produit, 1, true) -> "Commande : un " ^ produit
  | (produit, n, true) -> "Commande : " ^ (string_of_int
   ↪  n) ^ " " ^ produit ^ "s"
  | _ -> "En attente de confirmation"
;;
commande : string * int * bool -> string = <fun>
```

OCaml
```
commande ("livre", 0, false);;
- : string = "Panier vide."
commande ("livre", 1, false);;
- : string = "En attente de confirmation"
commande ("livre", 1, true);;
- : string = "Commande : un livre"
commande ("livre", 2, true);;
- : string = "Commande : 2 livres"
```

✔ Dès qu'une correspondance est trouvée avec un motif, les parties de l'expression `expr0` sont *liées aux noms figurant dans le motif.*

⚠

OCaml
```
let termine_par_z mot =
  let lettre_z = 'z' in
  let lettre_Z = 'Z' in
  match String.get mot 0 with
   | lettre_z -> true
   | lettre_Z -> true (* Warning: matching case
    ↪  unused. *)
   | _ -> false (* Warning: matching case unused. *)
;;
```

⚠

Un filtrage a lieu sur un motif : une « forme », pas sur une « variable ».

- `0` ou `'a'` sont des "formes" simples ;
- Un identifiant est une forme quelconque (une liaison locale est créée !).

OCaml
```
let ceci_est_la_fonction_constante_vraie x y =
  match y with
   | x -> true
   | _ -> false (* Warning: matching case unused. *)
;;
```

OCaml
```
let ceci_ne_compile_meme_pas x y =
  match (x, y) with
   | (x, x) -> true
   | _ -> false
;;
Error: Variable x is bound several times
```

2 Gardes

Définition 2.1

On dispose cependant d'une syntaxe de motifs conditionnels :

OCaml
```
| motif when expr_test -> expr
```

où `expr_test` est bien entendu de type `bool`.

OCaml
```
let egal x y =
  match y with
   | z when x = z -> true
   | _ -> false
;;
```

ou même

OCaml
```
let egal x y =
  match y with
   | _ when x = y -> true
   | _ -> false
;;
```

On perd complètement l'intérêt du `match ... with` dans cet exemple (assez artificiel) ! On verra des exemples intéressants prochainement.

⚠

Ne pas en abuser !