Compléments sur le "pattern matching"

Un exemple déjà vu.

```
# let rec fold_right f l a =
   match l with
     [] -> a
   | x::reste -> f x (fold_right f reste a);;
val fold_right:('a ->'b ->'b)->'a list->'b->'b =<fun>
# let somme l = fold_right (+) l 0;;
val somme : int list -> int = <fun>
# (+);;   --> plus préfixe
- : int -> int -> int = <fun>
```

Les différentes formes de motifs vues jusqu'ici sont les suivantes.

- Toute constante est un motif, étant entendu qu'une constante est soit un élément d'un type de base prédéfini (`bool`, `int`, `string`, ...), soit un constructeur d'arité 0, comme par exemple le constructeur `[]`).
- Toute variable est un motif.
- Si `C` est un constructeur (le constructeur binaire des listes `::`entre bien sûr dans cette catégorie même si, dans ce cas, la notation est infixe) d'arité $n \geq 1$ et si $p_1, \ldots, p_n$ sont des motifs, alors $p = \texttt{C}\ (p_1, \ldots, p_n)$ est un motif
- Si $l_1, \ldots, l_k$ sont certains des noms de champs d'un type enregistrement et si $p_1, \ldots, p_k$ sont des motifs, alors $p = \{l_1 = p_1; \ldots; l_k = p_k\}$ est un motif.

La construction `match` a la signification suivante. Si e, e_1,...,e_n sont des expressions et si p_1,...,p_n sont des motifs, alors

$$\texttt{match}\, e\, \texttt{with}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$$

est une expression, qui est équivalente à

$$(\texttt{function}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n)\, e$$

Inversement,

$$\texttt{function}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n$$

peut s'écrire

$$\texttt{function x} \rightarrow (\texttt{match x with}\, p_1 \rightarrow e_1 \mid \cdots \mid p_n \rightarrow e_n)$$

Pattern matching non exhaustif

```
# let f l = match l with
              [] -> 0;;

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::_
val f : 'a list -> int = <fun>
# f [];;
- : int = 0
# f [1];;
Uncaught exception: Match_failure ("", 10, 35).

# let (x::l)=[1;2;3];;

Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val l : int list = [2; 3]
```

Il existe un motif qui filtre n'importe quelle valeur : `_` . La valeur filtrée est perdue après le filtrage.

```
# let rec meme_lg l1 l2 =
    match (l1,l2) with
    ([],[]) -> true
  | (_::reste1, _::reste2) ->
              meme_lg reste1 reste2
  | (_, _) -> false;;
val meme_lg : 'a list -> 'b list -> bool = <fun>
# meme_lg [1] ["toto"];;
- : bool = true

# let nulle l =
   match l with
  [] -> true
 | _ -> false;;
val nulle : 'a list -> bool = <fun>
```

Autre écriture

```
# let nulle l = (l = []);;
val nulle : 'a list -> bool = <fun>
```

Un dernier point important sur les motifs : ils doivent être *linéaires*, c'est-à-dire qu'une même variable ne peut apparaître qu'au plus une fois dans un motif. Par exemple

```
# let eg = function (x,x) -> true | _ ->
false;;

This variable is bound several times in this matching
```

La construction `as` permet de nommer un sous-motif d'un motif.

```
# let f = function
    [] -> []
  | x::[] -> []
  | x::((y::m) as l) -> m@l;;
val f : 'a list -> 'a list = <fun>
```