Cours de programmation fonctionnelle et logique
Cours 2 : Listes et filtrage

Table des matières

1 Définition des listes

1.1 Différences entre les listes et les tableaux

- on n'accède pas directement à un élément d'une liste, il faut la parcourir séquentiellement, de sa tête vers sa queue ;
- les listes peuvent grossir dynamiquement quand la taille d'un tableau est fixée une fois pour toutes ;
- un élément ajouté à une liste est ajouté en tête de cette liste ;
- on ne peut pas changer la valeur d'un élément d'une liste.

1.2 Syntaxe

- la liste est délimitée par des crochets « [ ] » ;
- ses éléments sont séparés par des « ; ».

Liste composée des éléments 1, 2 et 3, dans l'ordre :

```
# [1;2;3];;
 - : int list = [1; 2; 3]
```

1.3 Deux constructeurs

- « `[]` » appelé `nil` (néant) et qui désigne la liste vide.
- « `: :` » appelé `cons`, pour constructeur de liste ; c'est l'opérateur qui rajoute un élément en tête d'une liste.

(la liste vide est *polymorphe*.)

```
# [];;
 - : 'a list = []
```

```
# 3::[];;
 - : int list = [3]
```

```
# 1::2::3::[];;
 - : int list = [1; 2; 3]
```

```
# 0::[1;2;3;];;
 - : int list = [0; 1; 2; 3]
```

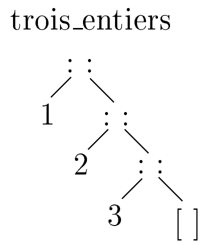
1.4 Représentation graphique des listes

```
# [e1;e2;e3;e4];;
```

est une abréviation pour

```
# e1::e2::e3::e4::[];;
```

```
# let trois_entiers = [1;2;3];;
 val trois_entiers : int list = [1; 2; 3]
```



2 Filtrage

Le filtrage est une **définition par cas**.

2.1 Filtrage par fonction anonyme

Suite de Fibonacci : $\texttt{Fib}(0) = 1$; $\texttt{Fib}(1) = 1$; $\texttt{Fib}(n+2) = \texttt{Fib}(n) + \texttt{Fib}(n+1)$.

```
# let rec fib n =
   if n=0
   then 1
   else if n=1
        then 1
        else (fib (n-2)) + (fib (n-1));;
           val fib : int -> int = <fun>
```

```
# let rec fib = function
     0 -> 1
   | 1 -> 1
   | n -> (fib (n-2)) + (fib (n-1));;
       val fib : int -> int = <fun>
```

Barre verticale « | » signifie « *ou* ». La flèche `->` indique l'expression à évaluer.

```
# let rec fib = function
     0 -> 1
   | 1 -> 1
   | n+2 -> (fib n) + (fib (n+1));;
       Characters 50-51:
Syntax error
```

Argument anonyme :

```
#let égal_un = function
    1 -> true
  | n -> false;;
     val égal_un : int -> bool = <fun>
```

```
#let égal_un = function
    1 -> true
  | _ -> false;;
     val égal_un : int -> bool = <fun>
```

Le symbole « _ » désigne *tous les autres cas.*

```
#let égal_un = function
    1 -> true;;
   Characters 14-35:
Warning: this pattern-matching is not exhaustive.
val égal_un : int -> bool = <fun>
```

```
#let égal_un = function
    _ -> false
  | 1 -> true;;
     Characters 40-41:
Warning: this match case is unused.
val égal_un : int -> bool = <fun>
```

2.2 Filtrage explicite

Calcul $(a/b) + c$: on veut vérifier que le deuxième argument (le diviseur) est non nul.

```
#let diventiere a = function
    0 -> (function c -> failwith "Division par zéro")
  | b -> (function c -> (a / b) + c);;
     val diventiere : int -> int -> int -> int = <fun>
```

```
#let diventiere a b c =
   match b with
    0 -> failwith "Division par zéro"
  | _ -> (a / b) + c;;
       val diventiere : int -> int -> int -> int = <fun>
```

`failwith` signale un cas d'erreur. Une fonction `Ocaml` doit toujours renvoyer une valeur, comme ici le calcul ne peut pas être effectué, on renvoie une **valeur exceptionnelle** signalant le problème.

2.3 Filtrage sur plusieurs arguments

Le filtrage ne peut se faire que sur **un seul argument** à la fois. Solution : l'utilisation de n-uplets.

```
#let pseudo_carré a b =
   match a with
     0 -> 0
   | _ -> match b with
           0 -> 0
          | _ -> a*a+b*b;;
          val pseudo_carré : int -> int -> int = <fun>
```

$$f(a,b) = \begin{cases} 0 & \text{si } a = 0 \\ 0 & \text{si } b = 0 \\ a^2 + b^2 & \text{sinon} \end{cases}$$

```
#let pseudo_carré a b =
   match (a,b) with
     (0,_) -> 0
```

```
| (_,0) -> 0
|    _    -> a*a+b*b;;
      val pseudo_carré : int -> int -> int = <fun>
```

3 Filtrage sur les listes

Fonction qui teste si une liste est vide.

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

```
#(nulle [], nulle [1]);;
 - : bool * bool = true, false
```

Nommage de la tête et du reste de la liste.

```
#let tete = function
   t::r -> t
 |  _   -> failwith "tete";;
     val tete : 'a list -> 'a = <fun>
```

Se lit : si la liste argument est non vide, on appelle `t` sa tête et `r` son reste.

```
#tete [1;2;3;4];;
 - : int = 1
```

```
#let reste = function
   _::r -> r
 |  _   -> failwith "reste";;
     val reste : 'a list -> 'a list = <fun>
```

```
#reste [1;2;3;4];;
 - : int list = [2; 3; 4]
```

Pour décomposer un type structuré, comme les listes, on filtre en utilisant les constructeurs du type, ici « [ ] » et « : : ».

```
#let bizarre = function
   []   -> 0
 | t::_ -> t;;
     val bizarre : int list -> int = <fun>
```

4 Synonyme dans les filtres

Ici, la paire (a, b) représentera le nombre $a * 10^b$.

```
#let valeur_absolue = function
   (a,b) -> if a < 0 then (-a,b) else (a,b);;
   val valeur_absolue : int * 'a -> int * 'a = <fun>
```

Dans le cas où a est positif, on aimerait pouvoir renvoyer directement la paire reçue en argument, ce que permet l'introduction d'un synonyme par `as` :

```
#let valeur_absolue = function
   (a,b) as paire -> if a > 0 then (-a,b) else paire;;
   val valeur_absolue : int * 'a -> int * 'a = <fun>
```