Récapitulatif fonctions :

```
fun x y -> x + y ;;
```

= fonction anonyme, peut prendre plusieurs arguments.

```
function
| 0 -> "zero"
| 1 -> "un seul"
| _ -> "plusieurs"
```

= fonction qui ne prend qu'un argument, et intègre le pattern matching.

exemple : `let rec map F = function | [] -> [] | e :: l -> F e :: map F l`

map a 2 arguments.

Polymorphisme : Ocaml utilise des variables de type (comme 'a, 'b, ...) on peut y mettre n'importe quel type.

```
fun x y = Some (x, y) ;;
```

Revoir Some, None, ...

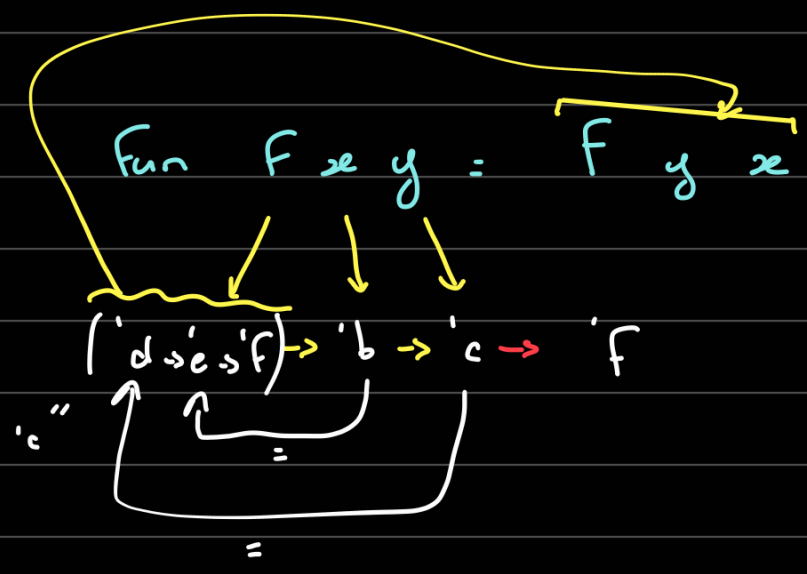
On peut forcer le type (à éviter car pas utile pour Ocaml)

```
fun (x : int) -> Some x ;;
```

↳ Restreint l'argument à un entier. Ce qui est dommage pour Ocaml

```
fun (x : 'a) -> Some x ;
```

↳ Inutile ici.



renvoie 'a

```
fun F -> F 0 ;;
```

int

(int -> 'a) -> 'a = <fun>

int = 'b

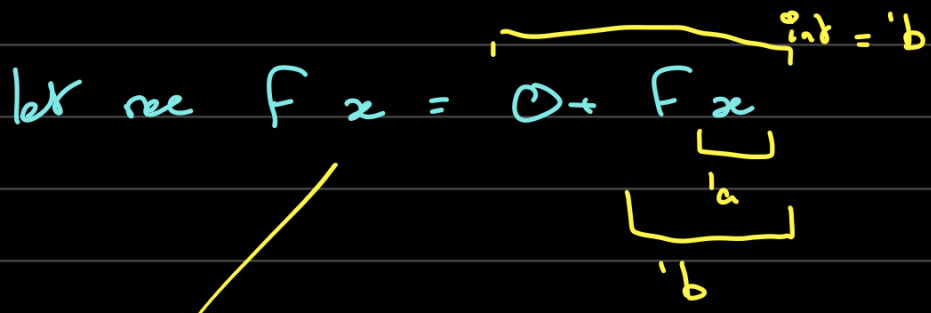
'a

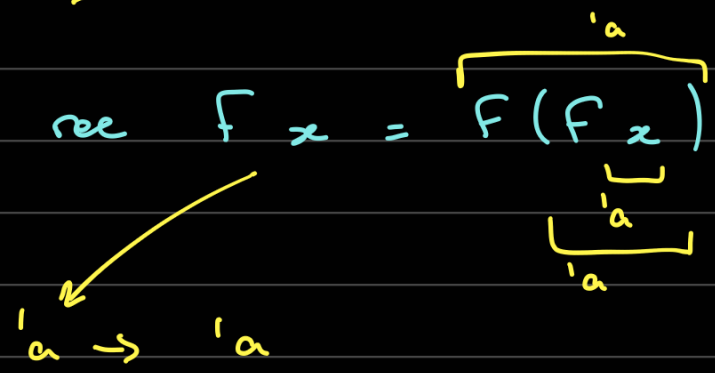
```
fun f x -> 0 + f x ;;
```

'b

('a -> int) -> 'a -> int

(x)





On peut écrire

```
let f (x, y) = x + y ;;
```

2-uplet

f n'est PAS une fonction à 2 arguments.

On le voit dans le top-level :

```
int * int -> int = <fun>
```

On doit appeler f comme ceci :

```
f (1,2)
```

(ce qui s'approche de Java, etc)

Style currifié :

```
let f p =
  match p with (x, y) -> x + y ;;
```

Style décurrifié :

```
let f (x, y) = x + y ;;
```

Le but est d'écrire les arguments successivement, pas sous forme de paires pour le style currifié.

Cela permet entre autre de faire des applications partielles plus simplement.

```
let incr = (+) 1;;
incr 10;;          ~> 11
```

→ Identique à `let incr x = x + 1;;`

```
List.map (fun x -> 2*x) [1;2;3];;
List.map ((*) 2) [1;2;3];;
```

exactement identique. → [2;4;6]

```
List.filter (fun x -> x < 5) [1;5;6];;
List.filter ((>) 5) [1;5;6];;
```

idem → [6]

⚠ On cherche $x < 5$ ⚠

car $(> 5\ x) \Leftrightarrow (5 > x)$