Introduction à la programmation fonctionnelle

Cours 5

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓
2 IPF (2) : fonctions récursives, inférence de types ✓
3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur ✓
4 IPF (4) : Exceptions, Listes (1) ✓
5 IPF (5) : Fonctions anonymes, Itérateurs, Listes (2)
- 5.1 Fonctions anonymes
- 5.2 Itérateurs avancés
- 5.3 Applications partielles

Les fonctions sont des valeurs comme les autres

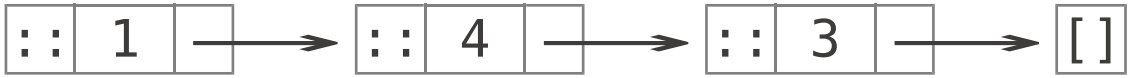
En programmation fonctionnelle lest fonctions sont des valeurs comme les autres :

- Les fonctions peuvent prises comme argument (ordre supérieur) ✓
- On doit pouvoir définir des fonctions « n'importe où »
- On doit pouvoir renvoyer des fonctions comme résultat

Qu'est-ce qu'une fonction ?

(on se pose la question du point de vue de la représentation en mémoire)

- Un entier : une suite de 64 bits
- Un flottant : une suite de 64 bits (organisée différemment)
- Une chaîne :un pointeur vers une zone mémoire. La zone mémoire contient un entier `t` (la taille de la chaîne) sur 64bits, puis `t` octets (les caractères)
- (un pointeur) :une suite de 64 bits
- Une liste : soit une valeur spéciale `[]` (correspondant au pointeur `null`) soit un pointeur vers un bloc de 2x64 bits : un pointeur vers la valeur dans la cellule, et un pointeur vers la suite



- une fonction ?

Représentation des fonctions

Considérons :

```
# let f =
    Printf.printf "Hello\n";
    let u = 42 in
    let g x = x + u in
    g;;
Hello
  val f : int -> int = <fun>
# f 1;;
- : int = 43
# u;;
Error: Unbound value u
```

- `f` est un « alias » pour `g`
- Lorsqu'on exécute `f` (donc `g`), `u` est bien visible et défini
- `g` « se souvient » de la variable `u`

Représentation des fonctions (2)

En mémoire, les fonctions sont représentées par des *clôtures*, c'est à dire un couple (c, e) où :

- c est un pointeur vers la zone mémoire contenant le code de la fonction
- e est un ensemble contenant les valeurs des variables non-locales à la fonction au moment de sa définition.

On peut stocker de tels objets dans des structures de données simplement (c'est juste un couple, on peut le mettre dans une liste, dans un autre couple, dans une variable, ...).

Lorsqu'on exécute une fonction, le processeur effectue un `call` (ou `jal`) vers l'adresse où se trouve le code. Avant ça il place e ainsi que tous les arguments sur la pile.

Lorsque le code accède à une variable non-locale, il va la chercher dans e

Fonctions anonymes

On a vu qu'on peut définir des fonctions partout avec la notation `let f x = ... in ...`

Si on reprend l'exemple de `List.iter` :

```
let pr_int x = Printf.printf "%d" x ;;
let pr_int_list l = List.iter print_int l ;;


let pr_int_list l =
   let pr_int x = Printf.printf "%d" x
   in
   List.iter pr_int l
;;
```

Dans le deuxième cas, on n'a pas polué les définition avec une fonction globale.

Est-ce qu'on peut écrire ça de façon plus simple ?

Fonctions anonymes (2)

Dans la définition précédente, on a défini une fonction pour ne l'utiliser qu'à un seul endroit :

```
let pr_int_list l =
   let pr_int x = Printf.printf "%d" x
   in
   List.iter pr_int l
;;
```

On peut ré-écrire le code comme ceci :

```
let pr_int_list l =
   List.iter (fun x -> Printf.printf "%d" x) l
;;
```

La notation `fun` x_1 … x_n`->` `e` permet de définir une fonction anonyme.

Fonctions anonymes (3)

Les fonctions anonymes sont particulièrement utiles lorsqu'elles sont utilisées avec des fonctions d'ordre supérieur:

```
let pr_int_list l =
  List.iter (fun x -> Printf.printf "%d" x) l
;;

let pr_float_list l =
  List.iter (fun x -> Printf.printf "%f" x) l
;;

let pr_int_int_list l =
  List.iter (fun x -> Printf.printf "<%d, %d>" (fst x) (snd x)) l
;;
```

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓
2 IPF (2) : fonctions récursives, inférence de types ✓
3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur ✓
4 IPF (4) : Exceptions, Listes (1) ✓
5 IPF (5) : Fonctions anonymes, Itérateurs, Listes (2)
- 5.1 Fonctions anonymes ✓
- 5.2 Itérateurs avancés
- 5.3 Applications partielles

le module `List`

En OCaml, les programmes sont structurés en modules. En particulier, chaque fichier définit un module. L'ensemble des fonctions et variables d'un module est accessible en utilisant le nom du module avec une Majuscule.

Le module `List` définit un grand nombre de fonctions utilitaires sur les listes.

le module `List` (2)

```
(* Ces trois fonctions sont à utiliser avec parcimonie ou pas du tout.
   On privilégiera le filtrage *)

List.length : 'a  list -> int   (* Longueur d'une liste *)
List.hd : 'a list -> 'a (* Première valeur. Lève une exception sur la liste
                           vide *)
List.tl : 'a list -> 'a (* Suite de la liste. Lève une exception sur la liste
                           vide *)
(* *)
List.append : 'a list -> 'a list -> 'a list (* Concatène deux listes. Peut aussi
                                               se noter l1 @ l2 *)
List.rev : 'a list -> 'a list (* Renverse une liste *)
```

Itérateurs de listes

Parmi les fonctions du module `List`, certaines sont des itérateurs. Elles permettent d'appliquer une fonction d'ordre supérieur à chaque élément d'une liste.

List.iter

C'est le premier itérateur que l'on a vu. Il permet d'appliquer une fonction qui ne renvoie pas de résultat à tous les éléments d'une liste. On l'utilisera principalement pour faire des affichages.

```
List.iter : ('a -> unit) -> 'a list -> unit
```

```
# List.iter (fun x -> Printf.printf "%b\n" x) [ true; false; true ] ;;
true
false
true
- : unit = ()
```

List.filter

Permet de filtrer, c'est à dire de renvoyer la liste de tous les éléments qui remplissent une certaine condition.

```
List.filter : ('a -> bool) -> 'a list -> 'a list
```

```
# List.filter (fun x -> x mod 2 = 0) [ 4; 5; 42; 1; 37; 49 ] ;;
  - : int list = [ 4; 42 ]
# List.filter (fun x -> x < 25.0) [ 10.5; 2.3; 99.0 ] ;;
  - : float list = [ 10.5; 2.3 ]
```

Le premier argument est appelé un prédicat.

List.map

Permet d'appliquer une transformation à chaque élément d'une liste et de renvoyer la liste des images.

`List.map f [` v_1 `; … ;` v_n `]` ⇝ [(f v_1); … ; (f v_n)]

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

```
# List.map (fun x -> x * x) [ 4; 8; 3 ] ;;
  - : int list = [ 16; 64; 9 ]
# List.map string_of_int [ 1; 2; 3 ] ;;
  - : string list = [ "1"; "2"; "3" ]
```

Ça va jusqu'ici ?
Ça va se corser un peu ...

Agrégations

On souhaite souvent « combiner » tous les éléments d'une liste. Exemple

$$\sum_{i=1..n} i^2 = (((1 + 4) + 9) + \ldots) + n^2$$

- Le $+$ est une opération binaire. On le rend « n-aire » en l'appliquant une première fois entre deux éléments. Puis en le répétant sur le résultat précédant et le nouvel élément, puis ...
- Quelle valeur renvoyer quand $n = 0$? 0 semble un choix raisonnable.

Ce type d'opération se retrouve souvent : somme, produit, min, max, ... :

$$\text{Min } \{ v_1, \ldots, v_n \} = \min(\min(\min (\min (v_1, v_2), v_3), \ldots), v_n)$$

Comment exprimer ce genre d'opérations par un opérateur générique ?

List.fold_left

Permet d'appliquer une fonction d'agrégation aux éléments d'une liste.

```
List.fold_left : ('a -> 'b -> 'a ) -> 'a -> 'b list -> 'a
```

La fonction prend trois arguments

- La fonction d'agrégation
- La valeur initiale (utilisée en particulier en cas de liste vide)
- La liste d'éléments

```
List.fold_left f a [ v1; …; vn ] ⇝ f (f (f (f (f a v1) v2) v3) …) vn
```

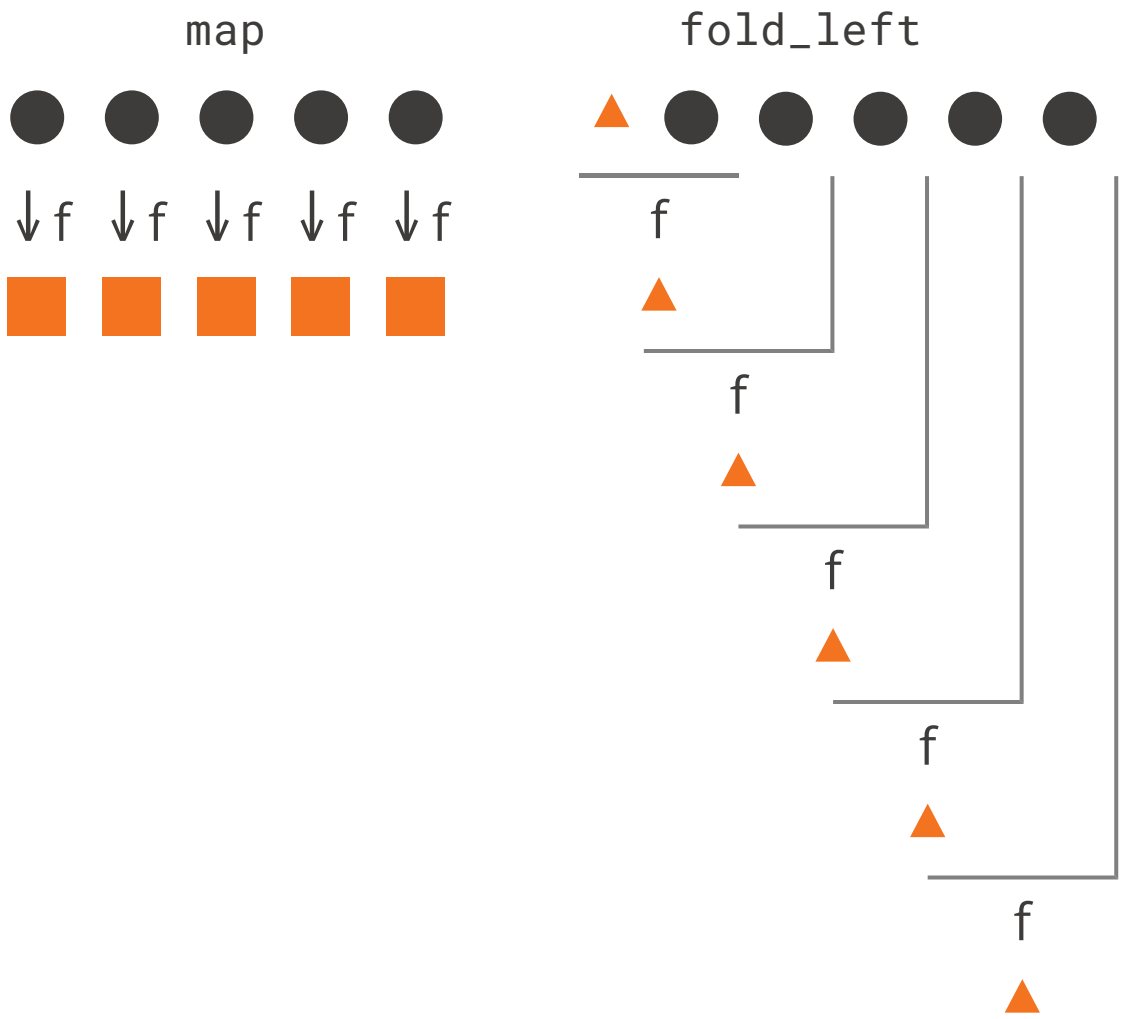
List.fold_left

```
# List.fold_left (fun a x -> a + x) 0 [ 1; 3; 7 ] ;;
  - : int = 11
# let f a x = a ^ " " ^ string_of_int x;;
  val f : string -> int -> string = <fun>
# List.fold_left f "" [ 1; 3; 7 ] ;;
- : string = "1 3 7 "
```

Dans le code ci-dessus:

```
List.fold_left f "" [ 1; 3; 7 ]
f (f (f "" 1) 3) 7
f (f  "1 " 3) 7
f "1 3 " 7
f "1 3 7 "
```

Visuellement



Tri d'une liste

À proprement parler, le tri d'une liste n'est pas un itérateur. Il utilise cependant de l'ordre supérieur.

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

La fonction `List.sort` prend en argument une fonction de comparaison.
La fonction de comparaison prend en argument deux valeur `a` et `b` et renvoie un nombre négatif (`a < b`), nul (`a = b`) ou positif (`a > b`).
En OCaml, la fonction prédéfinie `compare` a ce comportement.

```
compare : 'a -> 'a -> int
```

```
# List.sort compare [ 4; 8; 3 ] ;;
  - : int list = [ 3; 4; 8 ]
# List.sort compare [ "C"; "A"; "B"; "AX" ] ;;
  - : string list = [ "A"; "AX"; "B"; "C" ]
```

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓
2 IPF (2) : fonctions récursives, inférence de types ✓
3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur ✓
4 IPF (4) : Exceptions, Listes (1) ✓
5 IPF (5) : Fonctions anonymes, Itérateurs, Listes (2)
- 5.1 Fonctions anonymes ✓
- 5.2 Itérateurs avancés ✓
- 5.3 Applications partielles

Exemple

Considérons la fonction suivante :

```
let f x =
  (fun y -> x + y)
;;
```

C'est une fonction qui :

- prend en argument un entier `x`
- renvoie une fonction qui attend un `y` et renvoie `x + y`

```
# let f x =
  (fun y -> x + y)
;;
  val f : int -> int -> int = <fun>
```

Exemple (2)

```
# let g = f 3
  val g : int -> int = <fun>
# g 4;;
  - : int = 7
```

Ici, `f 3` renvoie une fonction (qu'on stocke dans `g`). Cette fonction attend un autre argument `y` et renvoie `3 + y`.

Si on s'intéresse aux types, quelle différence de type entre :

```
let f x = (fun y -> x + y) ;;
let add x y = x + y ;;
```

... aucune : `int -> int -> int`

Application partielle

On dit qu'une application est partielle si on applique une fonction à un nombre d'arguments inférieur à celui attendu.
En OCaml, ce n'est pas une erreur.
Si une fonction est de type : t_1 -> t_2 -> … -> t_n -> s, alors on peut l'appliquer à au plus n arguments.
Si on l'applique à k < n arguments, le résultat est une fonction qui attend les n-k arguments restant.

```
# let f x y z = x + y + z;;
  val f : int -> int -> int -> int  = <fun>
# let g = f 10;;
  val g : int -> int -> int  = <fun>
# let h = g 5;;
  val h : int -> int  = <fun>
# h 6;;
  - : int  = 21
```

Application partielle et ordre supérieur

L'application partielle est particulièrement utile, combinée à l'ordre supérieur

```
let pr_int_list = List.iter (Printf.printf "%d");;
(* int list -> unit *)

let incr_int_list = List.map ((+) 1);;
(* int list -> int list *)

let sum_list = List.fold_left (+) 0 ;;
(* int list -> int *)
```