Chapitre 0

Programmation fonctionnelle

Option informatique MP1, MP2 & MP3

Nous avons rencontré en première année des situations où on répétait le même type de construction qui utilisait une fonction et une liste.

L'exemple le plus courant est l'application d'une fonction aux éléments d'une liste :

```
let rec double liste =
  match liste with
  | [] -> []
  | t::q -> (2*t) :: (double q);;
```

```
let rec parite liste =
  match liste with
  | [] -> []
  | t::q -> (t mod 2 = 0) :: (parite q);;
```

Ce motif d'algorithme peut être factorisé à l'aide d'une méta-fonction dont les paramètres sont une fonction et une liste.

Nous allons définir quelques motifs et les méta-fonctions correspondantes.

Le code des fonctions est adapté de celui trouvé dans le fichier `list.ml`.

Les codes ne sont pas très compliqués et il est peut être plus utile de comprendre la structure utilisée plutôt que d'utiliser aveuglément les fonctions. En particulier, il est nécessaire de pouvoir traduire les idées utilisées ici dans le cas d'autres types récurifs comme les arbres ou les graphes.

I Application d'une fonction à une liste

On cherche à appliquer, MAPPER en anglais, une fonction à une liste :

Fonction map

```
let rec map f liste =
  match liste with
  | [] -> []
  | t::q -> (f t) :: (map f q);;

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

Les fonctions ci-dessus peuvent s'écrire

```
let double = List.map (fun x -> 2*x);;  
let parite = List.map (fun x -> x mod 2 = 0);;
```

II Itération d'une fonction sur une liste

Si une fonction a un résultat de type `unit`, on peut l'appliquer successivement à tous les éléments d'une liste,

```
Fonction iter  
let rec iter f liste =  
  match liste with  
  | [] -> ()  
  | t::q -> (f t);  
            iter f q;;  
  
iter : ('a ->unit) -> 'a list -> unit
```

C'est la structure semblable à une boucle `for` pour les listes.

Comme on n'autorise que des fonctions à résultat `unit`, on procède ici par effets de bords, ce n'est plus vraiment de la programmation fonctionnelle.

II.1 Impression de liste

```
let sortie k =  
  print_int k;  
  print_string " ";;  
  
let imprimer_liste l =  
  List.iter sortie l;  
  print_newline ();;
```

On ne peut pas remplacer la fonction `sortie` par une fonction anonyme (`fun x -> ...`) car il y a plusieurs instructions successives.

II.2 Somme des éléments d'une liste

```
let somme_liste l =  
  let s = ref 0 in  
  List.iter (fun x -> s := !s + x) l;  
  !s;;
```

III Itération cumulative d'une fonction sur une liste

Le calcul de la somme ci-dessus revient à écrire

```
let somme_liste l =
  let s = ref 0 in
  let rec aux a_faire =
    match a_faire with
    | [] -> ()
    | t::q -> s := !s + t;
              aux q in
  aux l;
  !s;;
```

Ce n'est pas un algorithme récursif lisible. On aurait plutôt écrit

```
let rec somme_liste l =
  match l with
  | [] -> 0
  | t::q -> t + somme_liste q;;
```

Pour une liste $[a_1; a_2; \dots; a_n]$, on a calculé $a_1 + (a_2 + (\dots + (a_n + 0) \dots))$.

En définissant `let f x y = x+y;;` on a calculé $f a_1 (f a_2 (\dots (f a_n 0) \dots))$.

De manière générale on retrouve souvent le calcul $f a_1 (f a_2 (\dots (f a_n b_0) \dots))$.

Fonction `fold_right`

```
let rec fold_right f liste b0 =
  match liste with
  | [] -> b0
  | t::q -> f t (fold_right f q b0);;

fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

III.1 Somme des éléments d'une liste

```
let somme_liste l =
  List.fold_right (fun x y -> x + y) l 0;;
```

III.2 Sucre syntactique

Un opérateur infixé comme `+`, `*`, ... peut être transformé en fonction de 2 variables en l'encadrant par des parenthèses : `(+)`, `(*)`,

III.3 Produit des termes d'une liste

```
let somme_liste l =
  List.fold_right (*) l 1;;
```

IV Autre itération cumulative

Une écriture récursive terminale de la somme des termes d'une liste peut être

```
let somme_liste l =
  let rec aux reste fait =
    match reste with
    | [] -> fait
    | t::q -> aux q (fait + t) in
  aux l 0;;
```

On a calculé $(\dots((0 + a_1) + a_2)\dots + a_n)$. De manière générale on retrouve souvent le calcul $f(\dots f(f a_0 b_1)b_2)\dots)b_n$; ici on avait $f = (+)$.

C'est ce que permet la meta-fonction

Fonction fold_left

```
let rec fold_left f accu liste =
  match liste with
  | [] -> accu
  | t::q -> fold_left f (f accu t) q;;

fold_right : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
```

Cette fonction est distincte de fold_right dans le cas où la fonction n'est pas associative.

IV.1 Somme des éléments d'une liste

```
let somme_liste l =
  List.fold_left (+) 0 l;;
```

IV.2 Conversion d'une liste en chaîne

```
let chaine l =
  List.fold_left (fun ch k -> ch^(string_of_int k) " " l);;
```

IV.3 Appartenance à une liste

```
let appartient k l =
  List.fold_left (fun rep t -> t=k || rep) false l;;
```

IV.4 Dernier élément d'une liste

```
let dernier l =
  match l with
  | [] -> None
  | t::q -> Some(List.fold_left (fun x y -> y) t q);;
```