Mini-cours de Programmation Fonctionnelle
avec OCaml

1 Pourquoi la programmation fonctionnelle ?

Un programme fonctionnel s'écrit comme une composition de **fonctions pures** : elles produisent un résultat à partir de leurs entrées, sans effet de bord (pas de mutation de variables, pas d'entrée/sortie mélangée au calcul). Cela rend le code plus **prévisible**, plus **facile à tester**, et naturellement **parallélisable**.

OCaml est un langage fonctionnel *strict* (évaluation eager) avec un système de types fort et inféré. Nous allons explorer les motifs de récursion fondamentaux qui apparaissent dans tout programme fonctionnel.

2 Récursion simple : la factorielle

Commençons par l'exemple classique :

```
let rec factorielle n =
  if n <= 1 then 1
  else n * factorielle (n - 1)
```

Le cas de base est $n \leq 1$ (on retourne 1). Le pas de récursion est $n \times \text{factorielle}(n-1)$. C'est une récursion **linéaire** : la pile d'appels grandit linéairement avec n.

3 Généralisation : le récurseur sur $\mathbb{N}$

La factorielle n'est qu'un cas particulier d'un motif général : la **récursion primitive** sur les entiers naturels. On peut l'extraire dans une fonction d'ordre supérieur :

```
let rec recurse n z f =
  if n = 0 then z
  else f n (recurse (n - 1) z f)
```

Signature : `recurse : int → 'a → (int → 'a → 'a) → 'a`

- `z` est la valeur de base (cas $n = 0$)
- `f n acc` est le pas de récursion : il reçoit l'entier courant `n` et le résultat de la récursion sur $n-1$

On récrit alors la factorielle comme une instance de `recurse` :

```
let factorielle n = recurse n 1 ( * )
```

Ici, `z = 1` ($0! = 1$) et `f n acc = n * acc`.

3.1 Autres exemples

```
(* Somme des n premiers entiers : 1 + 2 + ... + n *)
let somme n = recurse n 0 (+)

(* Puissance : x^n *)
let puissance x n = recurse n 1 (fun _ acc →x * acc)

(* Fibonacci : F_n *)
let fib n = fst (recurse n (0, 1)
                  (fun _ (a, b) →(b, a + b)))

(* Factorielle double : n!! *)
```

```
let factorielle_double n =
  recurse n 1 (fun k acc →
    if k mod 2 = n mod 2 then k * acc else acc)
```

Observation : `recurse` capture le motif « faire quelque chose avec l'entier k et le résultat du calcul sur $k-1$ ». C'est exactement le **fold** (ou **réduction**) sur l'intervalle $[1; n]$.

4 Générer des listes avec `recurse`

Puisque `recurse` paramètre le type de l'accumulateur (`'a`), on peut l'utiliser pour **construire des listes** :

```
(* Intervalle decroissant : [n; n-1; ...; 1] *)
let range_dec n = recurse n [] (fun k acc →k ::acc)

(* n copies d'une valeur *)
let replique n x = recurse n [] (fun _ acc →x ::acc)

(* Puissances de 2 : [2^n; 2^{n-1}; ...; 2] *)
let puiss2 n = recurse n [] (fun k acc →(1 lsl k) ::acc)
```

Pour obtenir l'ordre croissant, on peut renverser le résultat ou passer un état supplémentaire :

```
(* Intervalle croissant : [1; 2; ...; n] *)
let range n =
  List.rev (recurse n [] (fun k acc →k ::acc))
```

On peut aussi accumuler un état complexe (un **tuple**) pour construire des suites comme les factorielles ou les nombres de Fibonacci :

```
(* Liste des factorielles : [1!; 2!; ...; n!] *)
let factorielles n =
  let (_, lst) =
    recurse n (1, []) (fun k (fact, acc) →
      let fact' = fact * k in
      (fact', fact' ::acc))
  in
  List.rev lst
```

Idée-clé : `recurse` est un **fold** sur les entiers. Il transforme un entier n en une valeur de n'importe quel type (entier, liste, tuple, option…). C'est la raison de sa puissance.

5 Du récurseur sur les entiers au récurseur sur les listes

Le même schéma de pensée s'applique aux listes. Une liste est soit vide ($\square$), soit constituée d'une tête x et d'une queue xs. La récursion structurelle suit cette définition :

```
(* Produit des elements d'une liste *)
let rec produit lst =
  match lst with
  | [] →1
  | x ::xs →x * produit xs
```

C'est une instance de **fold_right** (ou **fold**) sur les listes :

```
let rec recurse_list f z lst =
  match lst with
  | [] →z
  | x ::xs →f x (recurse_list f z xs)
```

Signature : `recurse_list : ('a → 'b → 'b) → 'b → 'a list → 'b`

Les rôles sont les mêmes que pour `recurse` :

- `z` est la valeur pour la liste vide
- `f x acc` combine la tête `x` avec le résultat du calcul sur la queue

```
let produit2 = recurse_list ( * ) 1
let somme = recurse_list (+) 0
let longueur = recurse_list (fun _ acc →acc + 1) 0
```

6 Applications classiques de fold

6.1 Transformer une liste (map)

```
let map f lst =
  recurse_list (fun x acc →f x ::acc) [] lst
```

Applique `f` à chaque élément, préserve l'ordre.

6.2 Filtrer une liste (filter)

```
let filter p lst =
  recurse_list (fun x acc →
    if p x then x ::acc else acc) [] lst
```

Garde les éléments vérifiant le prédicat `p`.

6.3 Concaténer (append)

```
let concatene lst1 lst2 =
  recurse_list (fun x acc →x ::acc) lst2 lst1
```

6.4 Aplatir (flatten)

```
let aplatir lst =
  recurse_list (fun xs acc →
    recurse_list (fun x acc' →x ::acc') acc xs) [] lst
```

6.5 Maximum

```
let maximum lst =
  recurse_list (fun x acc →
    match acc with
    | None →Some x
    | Some m →Some (max x m)) None lst
```

> **Principe :** avec `recurse_list`, on écrit des fonctions qui **consomment** une liste pour produire une valeur. Les opérations de base (map, filter, fold) se combinent : un `map` suivi d'un `filter` est un simple `fold`.

7 fold_right vs fold_left

`recurse_list` est un **fold_right** : il parcourt la liste de droite à gauche (de la fin vers le début). L'ordre de traitement est celui de la récursion :

$$\text{fold\_right } f\ z\ [a_1; a_2; a_3] = f\ a_1\ (f\ a_2\ (f\ a_3\ z))$$

Il existe un pendant, **fold_left**, qui parcourt de gauche à droite :

$$\text{fold\_left } f\ z\ [a_1; a_2; a_3] = f\ (f\ (f\ z\ a_1)\ a_2)\ a_3$$

En OCaml :

```
let rec recurse_list_g f z lst =
  match lst with
  | [] →z
  | x ::xs →recurse_list_g f (f z x) xs
```

Signature : `recurse_list_g : ('b → 'a → 'b) → 'b → 'a list → 'b`

7.1 Quand utiliser l'un ou l'autre ?

- **fold_right** préserve l'ordre de la liste dans le résultat : `map`, `filter`, `concat`. Il est récursif (pile), donc dangereux sur de très grandes listes.
- **fold_left** est récursif terminal (pas de débordement de pile). Il accumule « en avant » : utile pour `prendre` les premiers éléments, `dedup`, `paquets`.

```
(* Prendre les n premiers elements (fold_left) *)
let prendre n lst =
  let (_, res) =
    recurse_list_g (fun (i, acc) x →
      if i < n then (i + 1, x ::acc)
      else (i + 1, acc)) (0, []) lst
  in
  renverse res
```

7.2 Tableau comparatif

Fonction	Variant	Ordre naturel
`somme / produit`	indifférent	– – –
`map / filter`	fold_right	préservé
`renverse`	fold_left	inversé
`prendre / dedup`	fold_left	gauche → droite
`maximum`	indifférent	– – –

8 Synthèse : une famille de récurseurs

Toutes les fonctions que nous avons vues suivent le même plan :

récurseur

$$\begin{cases} \text{sur } \mathbb{N} : & \texttt{recurse n z f} \\ \text{sur listes (droite)} : & \texttt{recurse\_list f z lst} \\ \text{sur listes (gauche)} : & \texttt{recurse\_list\_g f z lst} \end{cases}$$

Chaque récurseur est paramétré par :

1. une **valeur de base** (`z`) pour le cas vide / zéro

2. une **fonction de combinaison** (`f`) qui exprime comment ajouter un élément au résultat déjà calculé

Cette séparation permet de **réutiliser le motif de récursion** sans le réécrire. La factorielle, la somme, la puissance, la construction de listes, le parcours d'arbres... tous ces algorithmes deviennent des instances d'un même patron.

Exercices pour aller plus loin

1. Écrire `existe : ('a → bool) → 'a list → bool` avec `recurse_list`
2. Écrire `prefixe : int → 'a list → 'a list` avec `recurse_list_g` (prend les n premiers éléments)
3. Écrire `seuil : int → int list → int list` qui garde les éléments inférieurs à un seuil
4. Écrire la fonction `insere` qui insère un élément dans une liste triée, puis `tri_insertion` qui trie une liste par insertions successives

9 Conclusion

La programmation fonctionnelle consiste à :

- **Identifier les motifs de récursion** communs (fold sur les entiers, fold sur les listes, etc.)
- **Extraire ces motifs** en fonctions d'ordre supérieur
- **Réutiliser** ces combinateurs pour écrire des programmes plus courts, plus sûrs et plus faciles à raisonner

Les trois piliers sont : **récursion**, **polymorphisme**, **ordre supérieur**. Avec eux, on reconstruit la moitié de la bibliothèque standard en quelques lignes.

Ce cours a été généré à partir des échanges avec OpenCode sur les fichiers `factorielle.ml`, `test_factorielle.ml`, `exemples_recurse.ml`, `listes_recurse.ml` et `recurse_liste.ml`.