Cours 1

Programmation Fonctionnelle (en OCaml)

Introduction

Objectif de ce cours : découvrir la programmation fonctionnelle avec OCaml, comprendre pourquoi c'est important, et apprendre les bases.

Plan :

1. Histoire et importance d'OCaml
2. Développer un programme OCaml (REPL, compilation)
3. Fonctions : trois formes, variations, récursion
4. Listes : construction, pattern matching, récurseur
5. map, filter, fold...

La programmation fonctionnelle change votre façon de penser. Après OCaml, tous les autres langages vous semblerons bavards et fragiles.

Pourquoi OCaml est le plus important

Histoire : Né à l'INRIA (Caml $\rightarrow$ OCaml en 1996).
Ce qui le rend unique :

- **Typage fort + inférence** : le compilateur détecte les erreurs sans qu'on écrive les types
- **Pattern matching exhaustif** : un oubli $\rightarrow$ warning
- **Immutabilité par défaut** : une fonction = une fonction
- **Performance** : compilateur natif, proche du C
- **Industrie** : Jane Street (60M lignes), Facebook (Infer), Docker, Bloomberg
- **Langage du prouveur Coq** (preuve formelle)

Comment développer un programme OCaml

Trois manières :

1. **Toplevel (REPL)** : `ocaml`, tapez des expressions

```
# 1 + 2 * 3;;
- : int = 7
# let carre x = x * x;;
val carre : int →int = <fun>
```

2. **Script** : `ocaml prog.ml`

3. **Compilation** : `ocamlc -o prog prog.ml` ou `ocamlopt -o prog prog.ml` (natif)

▶ **À faire :** Ouvrez `ocaml`. Calculez 2^{10}, définissez `carre x = x * x`, testez `carre 12`.

Pourquoi la programmation fonctionnelle ?

- Programme = composition de fonctions pures
- Pas d'effets de bord : $f(2)$ retourne toujours le même résultat
- **Prévisible**, **facile à tester**, **parallélisable**

Une fonction pure, c'est comme un distributeur de bonbons : vous mettez une pièce, vous obtenez un bonbon. Si vous devez secouer la machine pour qu'elle marche, ce n'est plus une fonction pure.

▶ **À faire :** Pourquoi une fonction pure est-elle plus facile à tester qu'une fonction qui modifie une variable globale ?

Les trois formes de fonctions

1. Anonyme (lambda)

```
fun x →x * x
fun x y →x + y
```

2. Nommée

```
let carre x = x * x
let somme a b = a + b
```

3. Récursive : `let rec` permet de s'appeler soi-même

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

▶ **À faire :** Écrivez `max2 x y` (max de deux entiers), puis `max3 x y z` utilisant `max2` (sans la fonction `max` de la bibliothèque).

Fonctions : variations et application partielle

Plusieurs écritures, même résultat :

```
let add a b = a + b (* classique *)
let add = fun a b →a + b (* lambda *)
let add = (+) (* alias d'opérateur *)
```

Application partielle :

```
let add5 = add 5 (* val add5 : int →int *)
(* add5 3 →8 *)
```

Opérateurs personnalisés :

```
let ( *** ) x y = x * x + y * y (* 3 *** 4 →25 *)
```

▶ **À faire :** 1. Définissez `incr = (+) 1` et testez `incr 10`.
2. Créez `prefixe p n = p ^ " " ^ n` puis `dire_bonjour = prefixe "Bonjour"`.

La factorielle (récursion simple)

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

Cas de base : $n \leq 1 \rightarrow 1$ **Pas récursif :** $n \times \text{factorielle}(n-1)$

Évaluation de `factorielle 4` : $4 \times (3 \times (2 \times (1 \times 1))) = 24$

Problème : chaque appel empile $n \times \ldots$ sur la pile. $O(n)$ en $\Rightarrow$ débordement pour n grand.

▶ **À faire :** Écrivez la suite d'appels de `factorielle 6`. Combien d'appels récursifs ? Quelle complexité ?

Récursion terminale (tail rec)

Idée : accumulateur pour le résultat partiel.

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

let factorielle n = fact_tail n 1
```

Évaluation : `fact_tail 4 1` $\rightarrow$ `fact_tail 3 4` $\rightarrow$ `fact_tail 2 12` $\rightarrow$ `fact_tail 1 24` $\rightarrow$ `24`

Plus de pile à déplier : OCaml transforme en boucle. $O(n)$ en temps, $O(1)$ en pile.

▶ **À faire :** Convertissez `somme n = if n=0 then 0 else n + somme (n-1)` en version tail-récursive. Testez `somme 100`.

Le récurseur sur $\mathbb{N}$

On extrait le **motif** de la récursion :

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

`recurse` généralise **toute** récursion sur un entier :

```
let factorielle n = recurse n 1 ( * )
let somme n = recurse n 0 (+)
let puissance x n = recurse n 1 (fun _ acc →x * acc)
```

▶ **À faire :** Écrivez `sum_pair n` qui somme les entiers pairs de 0 à n avec `recurse`. Et si on veut les multiples de 3 ?

Suite de réécriture

`recurse 5 0 (+)` se déroule pas à pas :

$$
\begin{aligned}
&\texttt{recurse 5 0 (+)} \\
\rightarrow\ & 5 + \texttt{recurse 4 0 (+)} \\
\rightarrow\ & 5 + (4 + \texttt{recurse 3 0 (+)}) \\
\rightarrow\ & 5 + (4 + (3 + \texttt{recurse 2 0 (+)})) \\
\rightarrow\ & 5 + (4 + (3 + (2 + \texttt{recurse 1 0 (+)}))) \\
\rightarrow\ & 5 + (4 + (3 + (2 + (1 + \texttt{recurse 0 0 (+)})))) \\
\rightarrow\ & 5 + (4 + (3 + (2 + (1 + 0))))
\end{aligned}
$$

Déduction de la complexité :

- Chaque appel réduit n de 1 $\Rightarrow$ $n + 1$ appels
- Chaque appel fait $O(1)$ travail
- $\Rightarrow$ $O(n)$ en temps, $O(n)$ en pile

▶ **À faire :** Écrivez la suite de réécriture de `recurse 3 1 ( * )`. Quelle complexité ? Combien de multiplications ?

Listes en OCaml

Deux constructeurs :

- `[]` — liste vide
- `x :: xs` — ajoute `x` devant `xs` (cons)

Sucres syntaxiques :

```
[1; 2; 3] (* = 1 ::2 ::3 ::[] *)
[] (* liste vide *)
```

Pattern matching :

```
let rec somme lst = match lst with
  | [] →0
  | x ::xs →x + somme xs
```

▶ **À faire :** Écrivez `longueur` qui compte les éléments d'une liste. Testez `longueur [5;2;8]`.

Listes : exemples et variations

Version tail-récursive de `somme` :

```
let rec somme_tail lst acc = match lst with
  | [] →acc
  | x ::xs →somme_tail xs (x + acc)
```

Fonctions classiques :

```
let rec membre x lst = match lst with
  | [] →false
  | t ::_ when t = x →true
  | _ ::xs →membre x xs

let rec concat a b = match a with
  | [] →b
  | x ::xs →x ::concat xs b

let rec dernier lst = match lst with
  | [] →None
  | [x] →Some x
  | _ ::xs →dernier xs
```

▶ **À faire :** Écrivez `premier` (premier élément, `option`). Puis `reverse` qui inverse une liste.

Récurseur sur listes

Même schéma que pour les entiers :

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

C'est **fold_right** : on remplace `[]` par z, `::` par f.

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

▶ **À faire :** Écrivez `reverse` avec `recurse_list`. Indice : $f(x, acc) =$ concat acc $[x]$. Pourquoi est-ce lent ?

map, filter, concat, flatten

Toutes s'écrivent avec `recurse_list` :

```
let map f lst = (* [1;2;3] |>map (fun x ->x*2) *)
  recurse_list (fun x acc ->f x ::acc) [] lst
  (* ->[2;4;6] *)

let filter p lst = (* [1;2;3;4] |>filter (fun x ->x mod 2 = 0) *)
  recurse_list (fun x acc ->if p x then x ::acc else acc) [] lst
  (* ->[2;4] *)

let concatene a b = (* concatene [1;2] [3;4] ->[1;2;3;4] *)
  recurse_list (fun x acc ->x ::acc) b a

let aplatir lst = (* aplatir [[1;2];[3]] ->[1;2;3] *)
  recurse_list (fun xs acc ->
    recurse_list (fun x acc' -> x :: acc') acc xs) [] lst
```

▶ **À faire :** Écrivez `list_find p lst` qui retourne `Some x` pour le premier élément vérifiant p, `None` sinon.

fold_right vs fold_left

fold_right (non terminal) : $f\ a_1\ (f\ a_2\ (f\ a_3\ z))$

fold_left (terminal) : $f\ (f\ (f\ z\ a_1)\ a_2)\ a_3$

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

▶ **À faire :** Que donne `fold_left (fun acc x -> x :: acc) [] [1;2;3]` ? Et avec `recurse_list` (fold_right) ? Pourquoi la différence ?

Les 3 piliers

Récursion	Polymorphisme	Ordre supérieur
cas base + pas	`'a list`	fonctions qui prennent des fonctions
	pour tout type	

Avec ces trois-là, on reconstruit la moitié de la bibliothèque standard en quelques lignes. Sans transpirer.

Conclusion

Points clés du cours 1 :

- OCaml : typage fort, immuable, performant
- Fonctions : `fun`, `let`, `let rec`
- Récursion, tail recursion, récurseur sur $\mathbb{N}$
- Listes : `[]`, `::`, pattern matching
- `recurse_list` = `fold_right`
- map, filter, fold... en 2 lignes

Prochain cours (cours 2) : le mot-clé `type`, les types algébriques (sommes, produits, arbres, AST), le pattern matching exhaustif, la compilation en bytecode.

Si vous avez compris `recurse_list`*, vous avez compris la moitié d'OCaml. L'autre moitié, c'est* `type`*.*

Exercices

1. `map` avec `fold_left` ? Pourquoi est-ce que ça inverse la liste ?
2. `sum_square n` = somme des carrés $1^2 + 2^2 + \cdots + n^2$ avec `recurse`.
3. `filter` sans `recurse_list`, avec `match lst with [] → [] | x :: xs → ...`.
4. `max_list lst` qui retourne le plus grand élément (utiliser `recurse_list`).
5. Version tail-récursive de `reverse`.
6. Une fonction `range a b` qui génère la liste $[a; a+1; \ldots; b]$ avec `let rec`.
7. Avec `recurse`, implémentez `fib n` (Fibonacci) en $O(n)$.