Second cours de Programmation Fonctionnelle

« Ocaml n'est pas un animal de compagnie »

1 Introduction : pourquoi ce cours existe

Dans le premier cours, on a vu des motifs de récursion (fold, recurse, recurse_list...). C'était bien, mais on est passés très vite sur la syntaxe. Ce second cours reprend tout depuis le début avec **les types** comme fil rouge. On va construire nos propres briques (listes, dictionnaires, arbres, un processeur virtuel...) en comprenant chaque ligne.

> *La programmation fonctionnelle, c'est comme la cuisine : si vous ne comprenez pas la différence entre une carotte et une carotte coupée en dés, vous allez avoir du mal à suivre la recette. Les types, c'est pareil.*

2 Rappels syntaxiques express

Avant d'attaquer les types, revoyons les trois formes qui permettent de créer des fonctions.

2.1 `fun` — la fonction jetable

Une fonction anonyme s'écrit avec `fun` :

```
fun x →x * x (* carre *)
fun x →x mod 2 = 0 (* test de parite *)
fun x →[x] (* mettre dans une liste *)
fun x y →x + y (* deux parametres *)
```

On les utilise surtout en les passant à `List.map` ou `List.filter` :

```
List.map (fun x →x * x) [1;2;3;4;5]
List.filter (fun x →x mod 2 = 0) [1;2;3;4;5;6]
```

> *Les fonctions anonymes, c'est comme les tickets de caisse : on les utilise une fois et on les jette. Pratique, pas besoin de leur donner un nom.*

2.2 `let` — donner un nom

`let` permet de nommer une valeur ou une fonction :

```
let pi = 3.14159
let double x = 2 * x
(* Equivalent a : *)
let double = fun x →2 * x
```

Avec `let...in`, on crée des noms temporaires :

```
let aire_disque r =
  let pi = 3.14159 in
  pi *. r *. r
```

2.3 `let rec` — quand la fonction s'appelle elle-même

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

La récursion sur les listes utilise `match` pour suivre la structure de la donnée :

```
let rec longueur lst = match lst with
  | [] →0
  | _ ::xs →1 + longueur xs

let rec map f lst = match lst with
  | [] →[]
  | x ::xs →f x ::map f xs
```

3 Faites-les, vos opérateurs !

En OCaml, les opérateurs sont des fonctions comme les autres. On les définit entre parenthèses :

```
let (|>) x f = f x (* pipe : flot de donnees *)
```

> *Regardez bien cette ligne. C'est tout. Juste `f x`. L'opérateur `|>` qui fait vibrer les développeurs F# depuis des années est une fonction de deux lignes que vous auriez pu écrire vous-mêmes. Parfois, la puissance se cache dans la simplicité.*

Le pipe permet d'écrire le traitement comme un pipeline (à la Unix) :

```
(* Sans pipe : on lit de l'interieur vers l'exterieur *)
let r = List.map (fun x ->x*x)
        (List.filter (fun x ->x mod 2 = 0) [1..6])

(* Avec pipe : on lit de gauche a droite *)
let r =
  [1; 2; 3; 4; 5; 6]
  |>List.filter (fun x ->x mod 2 = 0)
  |>List.map (fun x ->x * x)
```

On peut aussi définir la composition :

```
let (>>) f g x = g (f x)
(* double_puis_carre = fun x ->(x*2)^2 *)
let double_puis_carre = (fun x ->2*x) >>(fun x ->x*x)
```

La priorité des opérateurs est déterminée par leur premier caractère. C'est fixe, on ne peut pas la choisir : `|>` a une priorité très basse, parfait pour chaîner.

4 Les types algébriques : votre première structure de données

Le mot-clé `type` permet de définir ses propres types. C'est là qu'OCaml devient vraiment intéressant.

4.1 Les entiers de Peano (ou « comment compter sur ses doigts en OCaml »)

```
type nat = Zero | Succ of nat
```

> *Un entier, c'est soit zéro, soit le successeur d'un entier. C'est comme ça que les mathématiciens grecs comptaient (enfin, s'ils avaient eu OCaml). Avouez que c'est plus clair que la notation indo-arabe.*

Chaque constructeur (`Zero`, `Succ`) peut être utilisé pour fabriquer des valeurs et les décomposer avec `match` :

```
let trois = Succ (Succ (Succ Zero))

let rec to_int n = match n with
  | Zero ->0
  | Succ n ->1 + to_int n
(* to_int trois = 3 *)

let rec add a b = match a with
  | Zero ->b
  | Succ n ->Succ (add n b)
(* add trois (Succ Zero) = 4 *)
```

Le pattern matching est la **seule** façon de récupérer la valeur contenue dans un constructeur. C'est un peu comme un GPS qui vous dit « si t'es à Zero, retourne b ; si t'es à Succ, enlève le Succ et continue ». On ne peut pas se tromper : le compilateur vérifie qu'on a traité tous les cas.

4.2 Les listes (version maison)

Les listes d'OCaml sont déjà intégrées, mais on peut les redéfinir nous-mêmes :

```
type 'a lst = Nil | Cons of 'a * 'a lst
```

> *`Nil` c'est la liste vide, `Cons` c'est un élément collé devant une autre liste. `Cons` vient de « construct », mais mes étudiants disent « Constantine » parce que ça colle à la tête.*

```
let rec longueur l = match l with
  | Nil →0
  | Cons (_, xs) →1 + longueur xs

let rec map f l = match l with
  | Nil →Nil
  | Cons (x, xs) →Cons (f x, map f xs)
```

Le `'a` dans `'a lst` veut dire « pour n'importe quel type ». C'est le **polymorphisme**. La même définition marche pour les listes d'entiers, de chaînes, ou d'expressions. Magique.

5 Dictionnaire : quand les listes deviennent intelligentes

Un dictionnaire (aussi appelé tableau associatif) stocke des paires clé-valeur. Avec notre type liste, c'est immédiat :

```
type ('k, 'v) dico = ('k * 'v) lst

let dico_vide = Nil
let dico_ajoute k v d = Cons ((k, v), d)
```

La recherche se fait en parcourant la liste :

```
let rec dico_cherche k d = match d with
  | Nil →None
  | Cons ((k', v), suite) →
    if k = k' then Some v
    else dico_cherche k suite
```

Notez le retour en `option` : si la clé n'existe pas, on retourne `None` plutôt que de planter. C'est plus sûr, et ça force l'appelant à gérer l'absence.

Avec le pipe `|>`, la construction devient élégante :

```
let d = dico_vide
  |>dico_ajoute "x" 10
  |>dico_ajoute "y" 20
  |>dico_ajoute "z" 30
```

> **Propriété mathématique amusante :** un dictionnaire est juste une liste. L'ordre d'insertion compte (la dernière clé prime), et la suppression coûte $O(n)$. C'est exactement comme un vrai bureau : le dossier du dessus cache celui du dessous.

6 Les arbres : une liste qui bifurque

Un arbre binaire est une structure où chaque nœud a deux sous-arbres :

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

On peut calculer sa hauteur :

```
let rec hauteur t = match t with
  | Leaf →0
  | Node (_, g, d) →1 + max (hauteur g) (hauteur d)
```

Ou le parcourir en ordre infixe :

```
let rec to_list t = match t with
  | Leaf →[]
  | Node (x, g, d) →to_list g @ [x] @ to_list d
```

> *Les arbres, c'est comme les listes, mais les nœuds ont deux enfants. Comme si vos parents avaient décidé d'avoir un deuxième enfant pour que la récursion soit plus intéressante.*

7 AST : votre programme devient une data structure

C'est ici que ça devient sérieux. On peut représenter une expression arithmétique comme un type OCaml :

```
type expr =
  | Int of int
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
```

> *C'est le genre de moment où l'on réalise qu'une addition, c'est juste un* `Add` *qui contient deux autres expressions. Votre programme est devenu une data structure.* Inception *en moins bien, mais en plus utile.*

On peut alors l'évaluer :

```
let rec eval e = match e with
  | Int n →n
  | Add (a, b) →eval a + eval b
  | Sub (a, b) →eval a - eval b
  | Mul (a, b) →eval a * eval b
```

Ou l'afficher (on dit *pretty-print*) :

```
let rec show e = match e with
  | Int n →string_of_int n
  | Add (a, b) →"(" ^ show a ^ " + " ^ show b ^ ")"
  | Sub (a, b) →"(" ^ show a ^ " - " ^ show b ^ ")"
  | Mul (a, b) →"(" ^ show a ^ " * " ^ show b ^ ")"
```

Mieux : on peut la **dériver formellement** comme au lycée !

```
let rec derive e = match e with
  | Int _ →Int 0
  | Add (a, b) →Add (derive a, derive b)
  | Mul (a, b) →Add (Mul (derive a, b), Mul (a, derive b))
```

C'est la règle de dérivation $(a \times b)' = a' \times b + a \times b'$, écrite en OCaml. Exactement comme dans un cours de maths, mais exécutable.

7.1 Ajout de variables

On étend le type avec un constructeur `Var` :

```
type expr_var =
  | Int of int
  | Var of string
  | Add of expr_var * expr_var
  | Sub of expr_var * expr_var
  | Mul of expr_var * expr_var
```

Et on utilise le dictionnaire pour donner une valeur aux variables :

```
let rec eval_var env e = match e with
  | Int n →Some n
  | Var x →dico_cherche x env
  | Add (a, b) →
    (match eval_var env a, eval_var env b with
     | Some va, Some vb →Some (va + vb)
     | _ →None)
  | Sub (a, b) →...
  | Mul (a, b) →...
```

> **Pourquoi `option` ?** Parce qu'une variable peut ne pas être définie. Avec `Some`/`None`, on force l'appelant à vérifier. Fini les `KeyError` surprises.

8 Bytecode et processeur virtuel

Maintenant, on compile nos expressions en bytecode, puis on exécute ce code sur un processeur virtuel. Comme un vrai compilateur, mais en miniature.

8.1 Bytecode symbolique

On définit des instructions postfixées (notation polonaise inversée) :

```
type instr =
  | PUSH of int
  | ADD | SUB | MUL
```

La compilation transforme l'arbre en liste plate d'instructions :

```
let rec compile e = match e with
  | Int n →[PUSH n]
  | Add (a, b) →compile a @ compile b @ [ADD]
  | Sub (a, b) →compile a @ compile b @ [SUB]
  | Mul (a, b) →compile a @ compile b @ [MUL]
```

> *`compile a @ compile b @ [ADD]`, c'est la traduction de « calcule a, calcule b, puis additionne ». C'est exactement ce que dirait un chef de chantier à ses ouvriers. Sauf qu'ici les ouvriers sont des entiers.*

8.2 Exécution à pile

Les instructions s'exécutent avec une pile (une simple liste) :

```
let exec code =
  let rec loop stack = function
    | [] →
      (match stack with [r] →r
       | _ →failwith "pile invalide")
    | PUSH n ::rest →loop (n ::stack) rest
    | ADD ::rest →
      (match stack with
```

```
      | a ::b ::s →loop (b + a ::s) rest
      | _ →failwith "ADD: pile sous-dimensionnee")
    | SUB ::rest →...
    | MUL ::rest →...
  in loop [] code
```

8.3 Code binaire et CPU

On encode chaque instruction en un entier (opcode) :

```
let assemble instrs =
  let rec go acc = function
    | [] →List.rev acc
    | PUSH n ::rest →go (n ::0 ::acc) rest
    | ADD ::rest →go (1 ::acc) rest
    | SUB ::rest →go (2 ::acc) rest
    | MUL ::rest →go (3 ::acc) rest
  in go [] instrs
```

Et on construit un processeur virtuel avec un pointeur d'instruction, une pile, et la mémoire de programme :

```
type cpu = {
  ip : int;
  stack : int list;
  code : int array;
}
```

Le cycle est simple : lire l'instruction à `code[ip]`, l'exécuter, avancer `ip`.

```
let rec run cpu =
  if cpu.ip >= Array.length cpu.code then
    match cpu.stack with
    | [r] →r
    | _ →failwith "pile invalide"
  else
    run (step cpu)
```

8.4 Trace pas-à-pas

Pour visualiser, on affiche l'état après chaque instruction :

```
let rec trace cpu = ...
```

Exemple de trace pour $(3 + 4) \times (10 - 2)$:

```
ip=0   PUSH 3    stack=[]
ip=2   PUSH 4    stack=[3]
ip=4   ADD       stack=[4; 3]
ip=5   PUSH 10   stack=[7]
ip=7   PUSH 2    stack=[10; 7]
ip=9   SUB       stack=[2; 10; 7]
ip=10  MUL       stack=[8; 7]
FINI   stack=[56]  => 56
```

On voit la pile grandir et rétrécir comme un yoyo. C'est ainsi que fonctionnent la plupart des vrais processeurs.

9 Synthèse : du type au processeur

Tout ce qu'on a fait suit une progression simple :

$$\text{Type algébrique} \rightarrow \text{Pattern matching} \rightarrow \text{Fonctions} \rightarrow \text{Bytecode} \rightarrow \text{CPU}$$

1. On **définit un type** (nat, lst, tree, expr)
2. On **pattern matche** dessus pour écrire des fonctions
3. On **compile** les structures en instructions plates
4. On **exécute** ces instructions sur une machine

C'est exactement comme ça que les vrais langages fonctionnent. La différence entre votre compilateur miniature et GCC, c'est juste quelques milliers de personnes et quelques décennies de travail.

10 Exercices

1. **Opérateur de choix :** définir l'opérateur `(<?>)` qui prend une valeur par défaut et une option, et retourne la valeur si `Some`, la valeur par défaut si `None`.
2. **AST étendu :** ajouter `Div` (division) et `Pow` (puissance, avec exposant entier) au type `expr`, puis étendre `eval`, `show` et `compile`.
3. **Simplification :** écrire une fonction `simplifie : expr -> expr` qui simplifie les expressions : $0 + e \rightarrow e$, $1 \times e \rightarrow e$, $e \times 0 \rightarrow 0$, etc.
4. **CPU avec méga-octet :** ajouter une mémoire vive (un tableau d'entiers) au processeur et une instruction `STORE addr` / `LOAD addr`.
5. **Arbre avec dictionnaire :** implémenter un arbre de recherche binaire (BST) où les clés sont des entiers, et les valeurs sont polymorphes.

Ce cours fait suite à `cours_fonctionnel.pdf` et s'appuie sur les fichiers `types_et_match.ml`, `operateurs.ml` et `bytecode.ml`.