Cours 2

Syntaxe et Types en OCaml

Introduction

Cours précédent (cours 1) :

- Fonctions : `fun`, `let`, `let rec`
- Récursion, tail recursion, récurseur sur $\mathbb{N}$
- Listes : `[]`, `::`, pattern matching
- `recurse_list` (fold_right), map, filter

Ce cours : le mot-clé `type`, les types algébriques, le pattern matching exhaustif, l'AST, la compilation en bytecode.

Le `type` en OCaml, c'est comme un plan pour Lego : vous décrivez les pièces et comment elles s'assemblent.

Les trois formes de fonctions (rappel)

- `fun x -> x * x` — anonyme
- `let f x = x * x` — nommée
- `let rec f n = ... f (n-1)` — récursive

▶ **À faire :** Écrivez une fonction `apply_twice f x` qui applique f deux fois : `apply_twice (fun x -> x * x) 2 = 16`.

Fonctions : variations et curryfication

Plusieurs écritures, même résultat :

```
let add a b = a + b
let add = fun a b →a + b
let add = fun a →fun b →a + b
```

Application partielle :

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

Pipe |> :

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

▶ **À faire :** 1. Créez `incr = (+) 1`. 2. Avec |>, écrivez `x |> f |> g` ↔ `g (f x)`.

Les types algébriques

Principe : on décrit les *formes* possibles d'une valeur. **Entiers de Peano :**

```
type nat = Zero | Succ of nat

let trois = Succ (Succ (Succ Zero))

let rec to_int n = match n with
  | Zero →0
  | Succ n →1 + to_int n

let rec add a b = match a with
  | Zero →b
  | Succ n →Succ (add n b)
```

▶ **À faire :** Écrivez `mult a b` qui multiplie deux `nat` en utilisant `add`. (Rappel : $a \times b = a + a + \cdots + a$)

Types sommes et produits

Type somme : OU logique (|)

```
type couleur = Rouge | Vert | Bleu
type option = Rien | Chose of int
```

Type produit : ET logique

```
type paire = int * string (* anonyme *)
type point = { x : int; y : int } (* record *)
```

Combinaison :

```
type forme = Cercle of float | Rect of float * float
type image = forme list (* une image = liste de formes *)
```

▶ **À faire :** Définissez un type `bool` maison (deux constructeurs) et implémentez `not`, `and_`, `or_`.

Les listes (version maison)

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

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

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

Le mot `Cons` *vient de construct, mais les étudiants disent Constantine parce que ça reste dans la tête.*

▶ **À faire :** Écrivez `filter p l` pour `'a lst` maison. Testez avec les couleurs `Rouge, Vert, Bleu`.

Dictionnaire = liste de paires

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

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
```

Le type `option` gère l'absence : pas de `KeyError`.

▶ **À faire :** Écrivez `dico_ajoute k v d` qui ajoute une entrée. Que faire si la clé existe déjà ?

Les arbres binaires

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

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

Les arbres, c'est comme les listes, mais avec deux enfants. Quand vos parents ont un deuxième enfant, la récursion devient plus intéressante.

▶ **À faire :** 1. Écrivez `taille t` (nombre de nœuds). 2. Écrivez `miroir t` (échange gauche/droite).

Arbres : parcours et variations

Trois parcours :

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

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

Arbre de recherche binaire (insertion) :

```
let rec ajoute x t = match t with
  | Leaf →Node (x, Leaf, Leaf)
  | Node (y, g, d) →
    if x < y then Node (y, ajoute x g, d)
    else Node (y, g, ajoute x d)
```

▶ **À faire :** Écrivez `recherche x t` qui teste si x est dans l'arbre ($O(\log n)$ si équilibré).

AST : votre programme devient une data structure

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

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
```

▶ **À faire :** Ajoutez `Div of expr * expr` au type `expr`. Que faire si le dénominateur est 0 ? (indice : `option`)

Dérivation formelle

Les règles de dérivation sont des **transformations d'AST** :

```
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))
```

$(a \times b)' = a' \times b + a \times b'$ écrite en OCaml ligne 5.

▶ **À faire :** 1. Ajoutez `Var of string` au type `expr`. 2. Mettez à jour `eval` et `derive`.

Bytecode et CPU virtuel

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

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]` = «calcule a, calcule b, additionne».

▶ **À faire :** Écrivez `exec prog` qui interprète les instructions sur une pile. (PUSH n empile n, ADD dépile deux valeurs et empile leur somme.)

Du type au processeur

1. **Définir un type** (forme des données)
2. **Pattern matcher** (tous les cas)
3. **Compiler** en instructions plates
4. **Exécuter** sur une machine

▶ **À faire :** Quel est le «type» de la pile du CPU ?

Conclusion

Points clés du cours 2 :

- `type` définit des formes de données (somme, produit)
- Pattern matching exhaustif : tous les cas doivent être couverts
- AST = le code source devient un type OCaml
- Compilation en bytecode = transformation d'AST en liste d'instructions

Prochain cours (cours 3) : les monades. On va généraliser le pattern `bind/return` qu'on a entrevu avec `option`.

Un type, c'est comme une boîte à outils. Une monade, c'est une boîte à outils qui range ses outils toute seule.

Exercices

1. Ajoutez `Div` à `expr`, `eval`, `compile`.

2. Écrivez un type `'a btree` (arbre binaire) avec les constructeurs `Feuille` et `Noeud`. ImplémenteZ `miroir` et `hauteur`.

3. Un type `rgb = R | V | B`. Écrivez `suivant : rgb -> rgb` qui cycle R $\rightarrow$ V $\rightarrow$ B $\rightarrow$ R.

4. `exec` du bytecode : implémenteZ la machine à pile.

5. Ajoutez `POW` au bytecode (a^b). Modifiez `compile` pour l'utiliser.

6. Généralisez `compile` pour supporter les variables : `type expr = ... | Var of string`.

7. Écrivez `simplifie : expr -> expr` qui simplifie $x + 0$, $x \times 1$, etc.