Cours 3

Monades en OCaml

Introduction

Cours précédent (cours 2) :

- Types algébriques, types sommes et produits
- Arbres binaires, AST
- Compilation en bytecode et machine à pile

Ce cours : le pattern `bind`/`return` appliqué à `option` (calculs qui échouent), `list` (non-déterminisme, requêtes), et `parsec` (analyse syntaxique).

Une monade, c'est trois ingrédients : un type, une opération `return` *qui met une valeur dans la boîte, et* `bind` *(*`>>=`*) qui enchaîne des calculs en gérant la boîte.*

Le type `option` (rappel)

`option` = `None` (absence) ou `Some x` (valeur).

```
let safe_div a b = if b = 0 then None else Some (a / b)
```

Problème : chaîner devient verbeux.

```
let f x = match safe_div 10 x with
  | None →None
  | Some y →match safe_div y 2 with
    | None →None
    | Some z →Some (z + 1)
```

▶ **À faire :** Combien de `match` pour 3 appels à `safe_div` ? Pour 10 ?

La monade `option` (bind/return)

```
let ( >>=) opt f = match opt with
  | None →None
  | Some x →f x

let retour x = Some x
```

»= (bind) chaîne les calculs :

```
let exemple =
  safe_div 10 2 >>=fun y →
  safe_div y 2 >>=fun z →
  retour (z + 1)
(* val exemple : int option = Some 3 *)
```

▶ **À faire :** Réécrivez `somme_opt` (somme d'une liste avec `option`) avec `»=` et `retour`.

Monade `option` : avant/après

Sans monade :

```
let f x y z = match safe_div x y with
  | None →None
  | Some a →match safe_div a z with
    | None →None
    | Some b →Some (b + 1)
```

Avec monade :

```
let f x y z =
  safe_div x y ≫=fun a →
  safe_div a z ≫=fun b →
  retour (b + 1)
```

▶ **À faire :** Que se passe-t-il si $y = 0$? $z = 0$? `f 10 0 2` retourne quoi ? Que se passe-t-il dans la version sans monade ?

La monade Liste

`'a list` = plusieurs résultats possibles (non-déterminisme).

```
let retour x = [x]

let ( >>=) m f = List.concat (List.map f m)
(* concat map : chaque élément peut produire 0, 1 ou N résultats *)
```

`retour` met une valeur dans une liste singleton. `bind` applique f à chaque élément, puis aplatit.

Le `bind` de la monade Liste, c'est comme un distributeur de chewing-gums qui, pour chaque chewing-gum, vous en redonne plusieurs. Vous finissez avec une montagne de chewing-gums.

▶ **À faire :** Calculez `[1;2;3] >>= fun x -> [x; -x]`. Combien d'éléments ?

Liste : produit cartésien

Toutes les paires possibles :

```
let cartesian xs ys =
  xs >>=fun x →
  ys >>=fun y →
  retour (x, y)
(* cartesian [1;2;3] ["a";"b"] =
   [(1,"a"); (1,"b"); (2,"a"); (2,"b"); (3,"a"); (3,"b")] *)
```

Chaque »= explose la liste en plusieurs branches. C'est le **non-déterminisme** : on explore toutes les combinaisons.

▶ **À faire :** Écrivez `triples xs ys zs` qui produit tous les triplets (x, y, z). Testez avec `[1;2] »= ....`

Liste : triplets pythagoriciens

$n \leq 10$, $a^2 + b^2 = c^2$:

```
let pythagore n =
  let r = List.init n (fun i →i + 1) in
  r ≫=fun a →
  r ≫=fun b →
  r ≫=fun c →
  if a * a + b * b = c * c then retour (a, b, c)
  else []
(* pour n = 10 →[(3,4,5); (4,3,5); (6,8,10); (8,6,10)] *)
```

[] = échec (on filtre), `retour` = succès. On traduit l'énoncé mathématique **littéralement**.

▶ **À faire :** Éliminez les doublons ($a \leq b$). Combien de triplets uniques pour $n = 20$?

Liste : parties d'un ensemble (power set)

Toutes les sous-listes possibles :

```
let rec parties = function
  | [] →[[]]
  | x ::xs →
    parties xs ≫=fun p →
    [p; x ::p]
(* parties [1;2;3] =
   [[]; [1]; [2]; [1;2]; [3]; [1;3]; [2;3]; [1;2;3]] *)
```

Chaque élément x double les possibilités : soit on le prend, soit on le laisse. La monade Liste gère l'explosion combinatoire.

▶ **À faire :** Combien de parties pour une liste de n éléments ? Programmez `parties_tail` (tail-rec).

Liste : langage de requêtes (SQL-like)

On manipule des tables avec la monade Liste.

```
type table = { entete : string list; lignes : string list list }

let where table pred =
  { table with lignes =
    table.lignes >>=fun ligne →
    if pred ligne then [ligne] else [] }

let select table noms = (* projection *)
  { entete = noms; lignes =
    table.lignes >>=fun ligne →
    [List.map (fun nom →...) noms] }
```

Chaque >>= est un `FROM` : on itère sur les lignes. `[]` = `WHERE` qui filtre. `retour` = `SELECT`.

▶ **À faire :** Écrivez la jointure : deux tables, `ville` en commun. >>= sur les deux tables.

Monade Parsec : analyse syntaxique

Problème : lire une chaîne, extraire une structure, gérer les erreurs.

```
type 'a parser = char list -> ('a * char list) option
(* [entrée] -> Some (résultat, reste) ou None *)
```

Le type dit tout : prend des caractères, retourne un résultat et le reste, ou échoue.

```
let retour x = (fun s -> Some (x, s))

let ( >>= ) p f =
  fun s -> match p s with
    | None -> None
    | Some (x, s') -> f x s'
```

▶ **À faire :** Quel est le type de `retour` pour un `int parser` ?

Parsec : parseurs de base

```
let item = function (* un caractère *)
  | [] →None
  | c ::s →Some (c, s)

let car c = (* un caractère précis *)
  item ≫=fun x →
  if x = c then retour x else None

let chiffre = (* un chiffre *)
  item ≫=fun c →
  if '0' <= c && c <= '9' then retour c
  else None
```

On combine les parseurs avec `>>=` :

```
car 'a' >>= fun _ -> car 'b' >>= fun _ -> chiffre
```

lit `ab3` et retourne `Some ('3', [])`.

▶ **À faire :** Écrivez `string s` qui parse une chaîne exacte. Indice : récursion sur la chaîne.

Parsec : combinateurs

```
let ( <|> ) p q = fun s →(* p ou q *)
  match p s with Some r →Some r | None →q s

let rec many p s = (* 0 ou plus *)
  match p s with
  | None →Some ([], s)
  | Some (x, s')␣→
␣␣␣␣many␣p␣s' ≫=fun xs →
    retour (x ::xs) s'
␣␣(*␣s' est passé àmany, pas s *)
```

Avec `many` et `chiffre` on parse un entier :

```
int = many chiffre »= fun cs -> retour (int_of_string cs)
```

▶ **À faire :** Écrivez `sep_by p sep` qui parse une séquence séparée par `sep`.

Des monades partout

Monade	Type	retour	»= (bind)
`option`	`'a option`	`Some x`	`None → skip`
`list`	`'a list`	`[x]`	`concat map`
`parsec`	`char list → ...`	`fun s → Some (x,s)`	`passe l'état`
`writer`	`'a * string list`	`(x,[])`	`concatène logs`
`IO`	`...`	`fun _ → x`	`séquence`
`async`	`Promise`	`Promise.resolve`	`.then()`

Le pattern bind/return est partout. Une fois vu, vous le verrez dans Promise.then, Optional.flatMap (Java), $_.then (jQuery)...

Conclusion

Points clés du cours 3 :

- Une monade encapsule des calculs avec contexte
- Deux opérateurs : `retour` + `>>=`
- `option` : élégance pour les calculs qui échouent
- `list` : non-déterminisme, requêtes, combinatoire
- `parsec` : analyse syntaxique par composition

Prochain cours (cours 4) : on utilise tout ça pour implémenter un mini-réseau de neurones en OCaml pur.

Les monades, c'est comme les lasagnes : des couches. Mais contrairement aux lasagnes, on peut les composer sans avoir mal au ventre.

Exercices

1. Avec la monade option, écrivez `div_list lst` qui divise chaque élément par le suivant, retourne `None` si division par 0.

2. `cartesian 3x` : produit cartésien d'une liste avec elle-même 3 fois.

3. `pythagore` sans doublons ($a \leq b$).

4. `parties` tail-recursive.

5. `string s` (parseur de chaîne exacte).

6. `sep_by p sep` (séquence séparée).

7. `int_list = int <|> string "(" >= ...` qui parse `"123"` ou `"(1,2,3)"`.