Troisième cours de Programmation Fonctionnelle

« Les monades, c'est comme les lasagnes : des couches »

1 Introduction

Les deux premiers cours nous ont appris à récurser dans tous les sens (factorielle, Fibonacci, listes, arbres...) puis à maîtriser la syntaxe, les types et les opérateurs. Il est temps de passer à la vitesse supérieure : **les monades**.

> *« Monade » est un mot qui fait peur. En réalité, une monade est juste un pattern qui revient tout le temps, comme le fait de devoir ranger sa chambre : c'est chiant, mais une fois qu'on a compris, tout est plus propre.*

Une monade, c'est trois ingrédients :

1. un **type** qui enveloppe une valeur (α monade) ;
2. une opération `return` qui met une valeur dans la monade ;
3. une opération `bind` (notée `>>=`) qui enchaîne des calculs monitorés par la monade.

On va voir deux monades très utiles : la **monade Liste** (pour le non-déterminisme et les requêtes) et la **monade Parsec** (pour l'analyse syntaxique).

2 La monade Liste

2.1 Définition

La monade Liste est sans doute la plus simple à comprendre : une valeur de type `'a list` représente plusieurs résultats possibles (ou aucun).

```
let return x = [x]

let (>>=) m f = List.concat (List.map f m)
```

> ***return*** *met une valeur dans une liste comme on met un chat dans une boîte : il y est tout seul, il peut en ressortir.* ***bind****, c'est le moment où on ouvre la boîte, on applique une fonction à son contenu, et on se retrouve avec* plein *de chats (ou de boîtes).*

2.2 Les trois lois (pour les puristes)

Une monade doit respecter trois lois. En pratique, la liste les vérifie :

```
(* 1. return x >>=f = f x *)
(* 2. m >>=return = m *)
(* 3. (m >>=f) >>=g = m >>=(fun x ->f x >>=g) *)
```

2.3 Produit cartésien

Le premier exemple classique : étant donné deux listes, produire toutes les paires possibles.

```
let cartesian xs ys =
  xs >>=fun x ->
  ys >>=fun y ->
  return (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 principe du **non-déterminisme** : on explore toutes les combinaisons.

> *Le produit cartésien, c'est comme les jeux de société où on doit choisir un personnage, une arme et un lieu.* `>>=` *est le maître du jeu qui dit « oui, tu peux essayer toutes les combinaisons, même le plombier avec le poisson dans la bibliothèque ».*

2.4 Triplets pythagoriciens

On cherche les triplets (a, b, c) avec $a^2 + b^2 = c^2$ et $a, b, c \leq n$. Avec la monade, c'est une traduction directe de l'énoncé :

```
let pythagore n =
  let range = List.init n (fun i ->i + 1) in
  range >>=fun a ->
  range >>=fun b ->
  range >>=fun c ->
  if a * a + b * b = c * c then return (a, b, c)
  else []
```

2.5 Parties d'un ensemble (power set)

Un classique de la récursion : 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]] *)
```

2.6 Chemins dans un graphe

Tous les chemins d'un nœud A à un nœud F dans un graphe orienté :

```
let rec chemins graphe depart arrivee =
  let rec aux courant visite =
    if courant = arrivee then return [courant]
    else if List.mem courant visite then []
    else
      voisins graphe courant >>=fun suivant ->
      aux suivant (courant ::visite) >>=fun chemin ->
      return (courant ::chemin)
  in
  aux depart []
(* Pour le graphe :
     A ->B ->D ->F
     A ->B ->E ->F
     A ->C ->E ->F
   on obtient les 3 chemins de A vers F *)
```

2.7 Un mini Taquin (Tic-Tac-Toe)

La monade Liste permet d'explorer tout l'arbre de jeu :

```
let coups_possibles plateau =
  List.init 9 (fun i ->i) >>=fun i ->
  match List.nth plateau i with
  | Vide ->return i | _ ->[]

let tous_les_coups plateau joueur =
  coups_possibles plateau >>=fun i ->
  return (jouer plateau i joueur)
```

> *À ce stade, la monade Liste explore tellement de possibilités qu'elle pourrait prédire votre avenir. Mais comme elle est pure, elle préfère rester fonctionnelle et vous laisser faire vos propres erreurs.*

3 Listes en compréhension = langage de requête

La monade Liste permet de faire mieux que des combinaisons mathématiques : elle permet d'écrire des **requêtes** sur des données, un peu comme du SQL.

3.1 Une table = une liste de listes

On représente une table par un en-tête (noms de colonnes) et une liste de lignes (chaque ligne est une liste de chaînes) :

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

Nos données : des employés avec nom, âge, ville, salaire.

```
let employes = {
  entete = ["Nom"; "Age"; "Ville"; "Salaire"];
  lignes = [
    ["Alice"; "25"; "Paris"; "45000"];
    ["Bob"; "30"; "Lyon"; "52000"];
    ["Charlie";"22"; "Paris"; "38000"];
    ["Diane"; "35"; "Marseille"; "60000"];
    ["Eve"; "28"; "Lyon"; "47000"];
    ["Franck"; "40"; "Paris"; "55000"];
    ["Grace"; "32"; "Toulouse";"49000"];
    ["Hugo"; "27"; "Marseille";"42000"];
  ]
}
```

3.2 SELECT et WHERE

Un SELECT projette sur certaines colonnes ; un WHERE filtre les lignes. Les deux s'écrivent naturellement avec `>>=` :

```
let select table noms =
  let indices = List.map (colonne_idx table) noms in
  let lignes = table.lignes >>=fun ligne →
    [List.map (fun i →List.nth ligne i) indices]
  in
  { entete = noms; lignes }

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

Usage :

```
(* Noms et salaires des employes de Paris *)
let q1 = select (where employes (fun l →
  valeur employes l "Ville" = "Paris")) ["Nom"; "Salaire"]
```

3.3 Agrégation : GROUP BY

Calcul du salaire moyen par ville :

```
let salaire_moyen_par_ville table =
  let villes = List.sort_uniq compare
    (List.map (fun l →valeur table l "Ville") table.lignes)
  in
  villes >>=fun v →
  let (total, count) = List.fold_left (fun (s,c) l →
    if valeur table l "Ville" = v then
      (s + int_of_string (valeur table l "Salaire"), c + 1)
    else (s, c)) (0, 0) table.lignes
```

```
  in
  return (v, string_of_int (total / count))
(* [("Lyon", 49500); ("Marseille", 51000);
    ("Paris", 46000); ("Toulouse", 49000)] *)
```

3.4 Jointure

On peut aussi faire des JOIN entre tables :

```
let jointure_ville t1 t2 =
  let new_entete = t1.entete @
    List.filter (fun c →not (List.mem c t1.entete)) t2.entete in
  let colonnes_t2 = ... in
  let new_lignes = t1.lignes ≫=fun l1 →
    t2.lignes ≫=fun l2 →
    if valeur t1 l1 "Ville" = valeur t2 l2 "Ville" then
      [l1 @ List.map ...]
    else []
  in
  { entete = new_entete; lignes = new_lignes }
```

> *Vous venez de réécrire SQL en 15 lignes d'OCaml. Félicitations, vous pouvez maintenant dire à vos amis que « les monades, c'est comme du SQL, mais en mieux ». Ils seront impressionnés (ou ils fuiront).*

4 La monade Parsec

4.1 Principe

Un **parseur** (ou analyseur syntaxique) lit une liste de caractères et produit un résultat accompagné du reste de l'entrée. Comme il peut y avoir plusieurs façons d'analyser (non-déterminisme), on retourne une **liste** de résultats :

```
type 'a parser = char list →('a * char list) list
```

C'est exactement la monade Liste ! `return` produit un succès sans rien consommer ; `bind` enchaîne deux parseurs.

```
let return x = fun input →[x, input]

let (≫=) p f = fun input →
  List.concat_map (fun (x, reste) →f x reste) (p input)
```

> *Les monades, c'est comme les poupées russes : la monade Parsec est construite sur la monade Liste, qui elle-même est construite sur… des listes. Et les listes, c'est juste des poupées russes qui ont arrêté de faire semblant.*

4.2 Les briques de base

```
(* Un caractère quelconque *)
let any = function [] →[] | c ::cs →[c, cs]

(* Un caractère vérifiant un prédicat *)
let satisfy p = any ≫=fun c →if p c then return c else fail

(* Un caractère précis *)
let char c = satisfy (fun x →x = c)

(* Une chaîne exacte *)
let string s = ...

(* Toujours échouer *)
let fail = fun _ →[]
```

4.3 Combinateurs

```
(* Alternative : essaie p1, puis p2 si p1 échoue *)
let alt p1 p2 = fun input →
  match p1 input with [] →p2 input | res →res

(* Répétition : zéro ou plus *)
let rec rep p =
  alt (p ≫=fun x →rep p ≫=fun xs →return (x ::xs))
      (return [])

(* Optionnel *)
let option p = alt (p ≫=fun x →return (Some x)) (return None)
```

4.4 Exemple : nombres

```
let digit = satisfy (fun c →c >= '0' && c <= '9')

let decimal : string parser =
  rep1 digit ≫=fun cs →
  return (String.of_seq (List.to_seq cs))
```

```
let entier : string parser =
  option (char '-') >>=fun sgn →
  decimal >>=fun d →
  return (match sgn with None →d | Some _ →"-" ^ d)
```

4.5 Exemple : identifiants

```
let lettre = satisfy (fun c →(c >= 'a' && c <= 'z')
                          || (c >= 'A' && c <= 'Z'))
let alphanum = lettre <|> digit

let identifiant : string parser =
  lettre >>=fun premiere →
  rep alphanum >>=fun suivantes →
  return (String.make 1 premiere ^
          String.of_seq (List.to_seq suivantes))
```

4.6 Exemple : expressions arithmétiques

On construit une grammaire hiérarchique pour respecter la précédence des opérateurs :

```
let addop = char '+' >>=fun _ →return (fun x y →Add (x,y))
and mulop = char '*' >>=fun _ →return (fun x y →Mul (x,y))

let rec expr input =
  (terme >>=fun x →
   rep (addop >>=fun op →
        terme >>=fun y →
        return (fun gauche →op gauche y)) >>=fun ops →
   return (List.fold_left (fun acc f →f acc) x ops)) input

and terme input =
  (facteur >>=fun x →
   rep (mulop >>=fun op →
        facteur >>=fun y →
        return (fun gauche →op gauche y)) >>=fun ops →
   return (List.fold_left (fun acc f →f acc) x ops)) input

and facteur input =
  (entier >>=fun n →return (Num n)) <|>
  (char '(' *> expr <* char ')') input
```

4.7 Exemple : CSV

Le CSV (Comma-Separated Values) se parse en deux coups de `sep_by` :

```
let cellule = rep (satisfy (fun c →c != ',' && c != '\n')) >>=...
let ligne = sep_by cellule (char ',')
let csv = sep_by ligne (char '\n')
```

> *Avec Parsec, on peut parser absolument tout. Les fichiers CSV, le JSON, le HTML (bon, pour le HTML, prévoyez un peu plus de* ***rep****), et même la liste de courses de votre colocataire (celle où il écrit « lait, œufs, truc-muche » sans préciser ce qu'est « truc-muche »).*

5 Calculatrice : parseur + évaluateur

5.1 AST (Arbre de Syntaxe Abstraite)

Avant de parser, on définit ce qu'on veut produire :

```
type expr =
  | Num of int
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Div of expr * expr
```

5.2 Parseur vers l'AST

On adapte le parseur d'expressions pour construire l'AST :

```
let addop = token (char '+' >>=fun _ ->
  return (fun x y ->Add (x, y))) <|>
           token (char '-' >>=fun _ ->
  return (fun x y ->Sub (x, y)))
and mulop = token (char '*' >>=fun _ ->
  return (fun x y ->Mul (x, y))) <|>
           token (char '/' >>=fun _ ->
  return (fun x y ->Div (x, y)))
```

5.3 Évaluateur

L'évaluateur parcourt l'AST et calcule le résultat :

```
let rec eval = function
  | Num n ->n
  | Add (g, d) ->eval g + eval d
  | Sub (g, d) ->eval g - eval d
  | Mul (g, d) ->eval g * eval d
  | Div (g, d) ->
    let d' = eval d in
    if d' = 0 then failwith "Division par zéro"
    else eval g / d'
```

5.4 REPL

La boucle interactive lit une ligne, la parse, l'évalue et affiche le résultat :

```
let rec repl () =
  Printf.printf "> %!";
  let ligne = input_line stdin in
  if ligne = "quit" || ligne = "exit" then ()
  else match parse ligne with
  | [] ->Printf.printf " Erreur de syntaxe\n%!"; repl ()
  | (ast, reste) ::_ ->
    Printf.printf " AST : %s\n%!" (affiche_ast ast);
    (try let r = eval ast in
        Printf.printf " = %d\n%!" r
     with Failure s ->Printf.printf " Erreur : %s\n%!" s);
    repl ()
```

Exemple d'exécution :

```
=== Calculatrice ===
> 1+2*3
  AST : (1 + (2 * 3))
  = 7
```

```
> (1+2)*3
  AST : ((1 + 2) * 3)
  = 9
> 10/0
  Erreur : Division par zéro
> --5
  AST : (0 - (0 - 5))
  = 5
```

Vous venez d'écrire une calculatrice. En 100 lignes. Sans une seule boucle `for` ni variable mutable. Votre prof de maths du lycée serait fier. Votre collègue développeur Java aurait déjà importé 15 bibliothèques pour faire la même chose.

6 Résumé : les monades en une page

Monade	Type	À quoi ça sert ?
Liste	`'a list`	Non-déterminisme, requêtes, exploration
Parsec	`char list → ('a * char list) list`	Analyse syntaxique
Option	`'a option`	Calculs qui peuvent échouer

Toutes ces monades partagent la même structure :

```
(* return : met une valeur dans le contexte monadique *)
val return : 'a →'a monade

(* bind : enchaîne deux calculs monadiques *)
val (≫=) : 'a monade →('a →'b monade) →'b monade
```

Et si vous avez tout compris, vous faites partie des 10% de la population mondiale capables d'expliquer ce qu'est une monade sans utiliser le mot « foncteur ». Les 90% restants continueront à faire du JavaScript en attendant que ça passe.

7 Exercices

1. **Toutes les permutations :** écrire une fonction `permutations : 'a list → 'a list list` qui retourne toutes les permutations d'une liste, en utilisant la monade Liste.
2. **SuDoku solver :** utiliser la monade Liste pour résoudre une grille de SuDoku 9×9 (remplir les cases vides en essayant tous les chiffres possibles).
3. **Calc avec priorités :** ajouter les opérateurs `%` (modulo) et `^` (puissance entière) à la calculatrice.
4. **Parseur de JSON minimalist :** écrire un parseur qui lit un sous-ensemble de JSON (objets, tableaux, chaînes, nombres) et retourne un AST `json`.
5. **Jointure multi-conditions :** étendre la fonction de jointure pour supporter des conditions autres que l'égalité (ex. : salaire $>$ 45000 ET ville $=$ Paris).
6. **Grammaire ambigüe :** modifier `alt` pour qu'il retourne *tous* les résultats (non-déterministe), et observer la différence sur `un_plus_plus`.

Ce cours fait suite à `cours_fonctionnel.pdf` et `cours_syntaxe.pdf`. Fichiers associés : `monade_liste.ml`, `parsec.ml`, `calculatrice.ml`.

A Annexe : autres monades avancées

A.1 Monade CPS (Continuation Passing Style)

La monade CPS capture l'idée de « ce qu'on fait ensuite » — la **continuation**. Chaque fonction reçoit un argument supplémentaire qui représente la suite du calcul. Le type est :

```
type ('a, 'r) cont = ('a →'r) →'r
```

Une valeur de type `('a, 'r) cont` est une fonction qui, étant donné une continuation `('a -> 'r)`, produit un résultat `'r`.

A.1.1 Définition

```
let return x = fun k →k x

let (≫=) m f = fun k →m (fun x →f x k)
```

> *CPS, c'est comme un restaurant où au lieu de commander et d'attendre votre plat, vous donnez au chef un numéro de téléphone pour qu'il vous appelle quand c'est prêt. Et le chef enchaîne les plats comme des continuations. Le problème, c'est quand le téléphone sonne dans le four.*

A.1.2 Exemple 1 : sortie anticipée (early return)

On peut simuler un `return` style impératif : une continuation qui court-circuite le reste du calcul.

```
(* Calcul avec sortie anticipee *)
let cherche_dans_liste pred lst =
  let rec aux = function
    | [] →None
    | x ::xs →
      if pred x then Some x
      else aux xs
  in
  aux lst
```

Avec CPS, on peut faire mieux :

```
(* Trouve le premier element pair et l'imprime *)
let premier_pair_cps lst =
  let ret = ref None in
  let k_finale x = !ret in
  let k_trouve x = ret := Some x; k_finale 42 in
  List.iter (fun x →
    if x mod 2 = 0 then k_trouve x
  ) lst;
  !ret
```

Bon, ce n'est pas encore très monadique. Voici un exemple plus parlant : la « coroutine » avec CPS.

A.1.3 Exemple 2 : generateur (coroutine)

Avec CPS, on peut suspendre et reprendre un calcul :

```
type ('a, 'r) gen = ('a →unit) →('a →'r) →'r

let yield x = fun return suspend →suspend x

let rec range_gen n =
  if n <= 0 then fun return _ →return ()
  else yield n ≫=fun _ →
       range_gen (n - 1)
```

A.1.4 Exemple 3 : amb (choix non-déterministe avec CPS)

L'opérateur `amb` (McCarthy) essaie toutes les valeurs possibles via CPS. C'est une version controlée du non-déterminisme :

```
(* amb : choisit une valeur parmi une liste *)
let amb lst = fun k_succ k_fail →
  let rec essayer = function
    | [] →k_fail ()
    | x ::xs →k_succ x (fun () →essayer xs)
  in
  essayer lst

(* Utilisation : trouver x,y,z dans [1..n] tels que x^2 + y^2 = z^2 *)
let triple_pythagoricien n =
  let range = List.init n (fun i →i + 1) in
  amb range ≫=fun x →
  amb range ≫=fun y →
  amb range ≫=fun z →
  if x * x + y * y = z * z then return (x, y, z)
  else fail
```

> *L'opérateur `amb` est le CCSD (Chef de la Choisissabilité Supervisée) du monde des monades : il explore toutes les possibilités et revient en arrière si ça foire. Comme quand vous essayez tous les snacks du distributeur avant d'admettre que vous voulez juste un Mars.*

A.2 Monade de Backpropagation (Différentiation Automatique)

La monade de backpropagation permet de calculer automatiquement les dérivées d'une fonction. On va présenter deux approches : la différentiation automatique en mode **forward** (nombres duaux) et en mode **reverse** (backpropagation).

A.2.1 Nombres duaux (mode forward)

Un nombre dual associe à une valeur v sa dérivée dv :

```
type 'a dual = { v : 'a; d : 'a }
```

Les opérations arithmétiques propageant la dérivée via la règle de chaîne :

```
let add a b = { v = a.v + b.v; d = a.d + b.d }
let mul a b = { v = a.v * b.v; d = a.v * b.d + b.v * a.d }
let sin a = { v = sin a.v; d = cos a.v *. a.d }
```

On peut en faire une monade si on veut chaîner des calculs avec dépendances :

```
type 'a ad = { value : 'a; deriv : 'a; mutable tape : ('a ad * 'a ad) option }

let return x = { value = x; deriv = 0.0; tape = None }

let (>>=) m f =
  let r = f m.value in
  r.deriv <- m.deriv +. r.deriv;
  r
```

Cette présentation est simplifiée. L'implémentation réelle stocke un graphe de calcul (Wengert list / tape).

> *La différentiation automatique, c'est comme se rappeler tous les couloirs qu'on a empruntés dans un labyrinthe, mais à l'envers. Le mode forward, c'est votre GPS qui recalcule le chemin à chaque intersection. Le mode reverse, c'est le chauffeur Uber qui vous demande « par où on est passés déjà ? » à la fin du trajet.*

A.2.2 Mode reverse (backpropagation) : une tape monadique

Le mode reverse est plus efficace pour les fonctions avec beaucoup de paramètres (réseaux de neurones). On enregistre toutes les opérations sur une **tape**, puis on la parcourt à l'envers pour propager les gradients.

```
type tape =
  | Leaf of { mutable grad : float }
  | Add of node * node
  | Mul of node * node
  | Sin of node
  | Cos of node
and node = { value : float; tape : tape }

(* Construction du graphe *)
let add a b =
  let r = { value = a.value +. b.value;
            tape = Add (a, b) } in
  r

let mul a b =
  let r = { value = a.value *. b.value;
            tape = Mul (a, b) } in
  r

(* Backpropagation : derivee partielle *)
let rec grad_node n =
  match n.tape with
  | Leaf _ ->1.0
  | Add (a, b) ->grad_node a +. grad_node b
  | Mul (a, b) ->
    grad_node a *. b.value +. a.value *. grad_node b
```

```
  | Sin a →cos a.value *. grad_node a
  | Cos a →-.sin a.value *. grad_node a
```

On peut encapsuler `node` dans une monade pour chaîner naturellement :

```
type 'a backprop = unit →node

let return x () = { value = x; tape = Leaf { grad = 0.0 } }

let (>>=) m f () =
  let a = m () in
  (f a.value) ()
```

Exemple d'utilisation :

```
let f x = sin (mul x x) (* f(x) = sin(x^2) *)

let x = { value = 2.0; tape = Leaf { grad = 0.0 } }
let y = f x (* calcule sin(4) *)
let dy = grad_node y (* derivee : cos(4) * 2*2 *)
```

Avec la monade de backpropagation, vous pouvez dériver n'importe quelle fonction sans faire une seule ligne de calcul différentiel. C'est comme tricher à un examen de maths avec une calculatrice quantique. Vos profs de prépa vous détesteraient, mais vos réseaux de neurones vous adoreront.

A.3 Mini interprète Prolog

Le Prolog est un langage de programmation **logique** : on énonce des faits et des règles, et le moteur cherche les solutions par **unification** et **backtracking**. La monade Liste est parfaite pour modéliser le backtracking !

A.3.1 Terms et substitution

```
type term =
  | Var of string
  | Const of string
  | Fun of string * term list

type substitution = (string * term) list
```

A.3.2 Unification

L'unification trouve la substitution la plus générale qui rend deux termes égaux :

```
let rec unifie t1 t2 subst =
  match t1, t2 with
  | Var x, _ →unifie_var x t2 subst
  | _, Var y →unifie_var y t1 subst
  | Const a, Const b when a = b →return subst
  | Fun (f, args1), Fun (g, args2) when f = g →
    unifie_liste args1 args2 subst
  | _ →fail
```

A.3.3 Programme logique

Un programme est une liste de clauses (faits ou règles) :

```
type clause = { tete : term; corps : term list }

type programme = clause list
```

Le moteur d'inférence cherche à prouver un but en essayant toutes les clauses :

```
let rec prouve programme buts subst =
  match buts with
  | [] →return subst
  | but ::reste →
    programme ≫=fun clause →
    (* Renommer les variables de la clause *)
    let clause' = renomme clause in
    unifie but clause'.tete subst ≫=fun subst' →
    prouve programme (clause'.corps @ reste) subst'
```

A.3.4 Exemple : généalogie

```
let prog = [
  (* parent(x,y) : x est parent de y *)
  fait "parent" ["jean"; "marie"];
  fait "parent" ["marie"; "paul"];
  fait "parent" ["paul"; "sophie"];
  (* grandparent(x,y) :- parent(x,z), parent(z,y) *)
  regle "grandparent" ["x"; "y"]
        [atome "parent" ["x"; "z"]; atome "parent" ["z"; "y"]];
  (* ancetre(x,y) :- parent(x,y) *)
  regle "ancetre" ["x"; "y"] [atome "parent" ["x"; "y"]];
  (* ancetre(x,y) :- parent(x,z), ancetre(z,y) *)
  regle "ancetre" ["x"; "y"]
        [atome "parent" ["x"; "z"]; atome "ancetre" ["z"; "y"]];
]
```

```
(* Requete : ancetre(jean, X) ? *)
(* Resultat : X = marie, X = paul, X = sophie *)
```

Quand on lance la requête `ancetre(jean, X)`, le moteur :

1. Essaie la première règle `grandparent` : échec (les noms ne correspondent pas).
2. Essaie la règle `ancetre` avec `parent(jean,z)` : trouve `z=marie`.
3. Puis `ancetre(marie, X)` : récursivement, trouve `X=paul`, puis `X=sophie`.
4. La monade Liste collecte toutes les solutions.

Prolog, c'est comme un détective privé qui explore toutes les pistes en même temps. La monade Liste, c'est son tableau d'enquête avec des fiches de toutes les couleurs. Le backtracking, c'est quand il réalise que le majordome avait un alibi et qu'il défait tout le fil rouge pour suivre la piste du jardinier.

A.3.5 Code complet du mini Prolog

Voici le squelette complet (env. 60 lignes) :

```
type term = Var of string | Const of string | Fun of string * term list
type substitution = (string * term) list
type clause = { tete : term; corps : term list }
type programme = clause list

(* Variables fraiches *)
let compteur = ref 0
let fraiche _ = compteur := !compteur + 1; Var ("_" ^ string_of_int !compteur)

let renomme clause =
  let rec aux = function
    | Var x →fraiche ()
    | Const _ as t →t
    | Fun (f, args) →Fun (f, List.map aux args)
  in
  { tete = aux clause.tete; corps = List.map aux clause.corps }

(* Unification *)
let rec unifie t1 t2 subst = match t1, t2 with
  | Var x, _ →(try unifie (List.assoc x subst) t2 subst
               with Not_found →return ((x, t2) ::subst))
  | _, Var y →unifie t2 t1 subst
  | Const a, Const b when a = b →return subst
  | Fun (f, a1), Fun (g, a2) when f = g →
    (try List.fold_left2 (fun s t1 t2 →
       s ≫=fun s' →unifie t1 t2 s') (return subst) a1 a2
     with Invalid_argument _ →fail)
  | _ →fail

and fail = fun _ →[]

(* Moteur d'inference *)
let rec prouve prog buts subst = match buts with
  | [] →return subst
  | but ::reste →
    prog ≫=fun clause →
    let clause' = renomme clause in
    unifie but clause'.tete subst ≫=fun subst' →
    prouve prog (clause'.corps @ reste) subst'

(* Fonctions auxiliaires *)
let const n = Const n
let var n = Var n
let atome nom args = Fun (nom, List.map (fun n →Const n) args)
let fait nom args = { tete = atome nom args; corps = [] }
```

```
let regle nom args corps = { tete = atome nom args; corps }
```

Vous venez d'écrire Prolog en 60 lignes d'OCaml. C'est environ 60 fois moins de lignes que le manuel d'utilisation de SWI-Prolog. Et ça compile du premier coup (enfin, après quelques tentatives).

A.4 Aller plus loin

Les monades présentées ici ne sont que la partie émergée de l'iceberg. Il en existe des dizaines d'autres :

- **State monad** : threader un état à travers des calculs (compteur, générateur d'ID, etc.).
- **IO monad** : gérer les entrées-sorties de manière pure (utilisée en Haskell).
- **Writer monad** : accumuler des logs ou des traces.
- **Reader monad** : partager un environnement de configuration.
- **Free monad** : construire des DSL (Domain Specific Languages) de manière modulaire.
- **Probability monad** : modéliser des distributions de probabilités (Monte Carlo, inférence bayésienne).

On dit qu'un développeur fonctionnel qui maîtrise les monades peut expliquer le changement de couche à un bébé. On dit aussi qu'il peut être dangereusement productif et faire en une après-midi ce qu'une équipe de 10 développeurs Java fait en une semaine. Mais ça, c'est peut-être une légende urbaine.

Cette annexe complète le `cours_monade.pdf`. Pour aller plus loin : le livre *Real World OCaml*, la série d'articles *You Could Have Invented Monads!* (et *And Then You've Actually Invented Them*), et le site `https://learn-ocaml.org/`.