Programmation fonctionnelle

Poly d'accompagnement du cours

Philippe Roussille

3iL 2A 2026

1 Introduction

Ce document accompagne le cours et les travaux pratiques de **programmation fonctionnelle**.

Il ne s'agit pas d'un manuel de syntaxe F#, mais d'un **support de raisonnement**.
L'objectif est de comprendre **ce que change réellement** la programmation fonctionnelle dans la manière de concevoir, lire et écrire du code.

Ce cours vise avant tout un **changement de posture mentale** : - passer d'un programme vu comme une suite d'actions, - à un programme vu comme une **transformation de données**.

Les exemples sont majoritairement en **F#**, mais les idées sont **directement transférables vers Java moderne** (lambdas, Streams).

2 Le bagage à vider

La programmation fonctionnelle est un **paradigme**, c'est-à-dire une manière structurée de penser les programmes.

Elle remet en question plusieurs réflexes courants :

- penser un programme comme une suite d'instructions,
- modifier des variables au fil du temps,
- raisonner avec de l'état global,
- utiliser l'héritage comme abstraction principale,
- gérer l'absence avec `null`,
- considérer les boucles comme la structure naturelle de tout calcul.

Ces réflexes ne sont pas faux, mais ils rendent le raisonnement difficile lorsque les programmes deviennent complexes.

3 Programme comme suite d'actions : un réflexe à questionner

Dans le paradigme impératif, un programme est souvent conçu comme :

— une séquence d'instructions,
— exécutées dans un ordre précis,
— modifiant un état au fil du temps.

Pour comprendre un tel programme, il faut :

— simuler l'exécution,
— suivre les mutations,
— garder en mémoire l'état courant.

Ce raisonnement est **temporel**, souvent **global**, et fragile face aux évolutions.

4 Changement de modèle mental

4.1 Programme = expression

En programmation fonctionnelle, un programme est vu comme une **expression**.

Une expression :

— décrit un calcul,
— produit une valeur,
— peut être remplacée par son résultat.

En F#, de nombreuses constructions sont des expressions :

```
let x = if a > b then a else b
```

Ici, le `if` produit une valeur. Il n'existe pas de branche "qui ne renvoie rien".

4.2 Fonction = unité de raisonnement

Une fonction n'est pas une procédure qui agit sur un état global. C'est une **transformation de données**.

```
let carre x = x * x
```

Cette fonction :

— dépend uniquement de son paramètre,
— ne modifie rien à l'extérieur,
— est compréhensible isolément.

Le raisonnement devient **local** et fiable.

5 Fonctions pures et effets de bord

5.1 Fonctions pures

Une fonction pure :

— dépend uniquement de ses paramètres,
— renvoie toujours le même résultat pour les mêmes entrées,
— ne produit aucun effet observable à l'extérieur.

```
let somme a b = a + b
```

Une fonction pure peut être lue comme une **équation mathématique**.

5.2 Effets de bord

Un effet de bord est toute interaction avec l'extérieur :

— affichage,
— lecture de fichier,
— génération aléatoire,
— dépendance à l'heure.

```
let afficher x =
    printfn "%d" x
```

Cette fonction a un effet de bord (affichage). L'objectif n'est pas de les supprimer, mais de les **isoler**.

6 Valeurs et immutabilité

En programmation fonctionnelle, on raisonne avec des **valeurs immuables**.

```
let x = 10
let y = x + 1
```

Ici, `x` ne change jamais. On calcule une **nouvelle valeur** au lieu de modifier l'ancienne.

L'immutabilité :

— simplifie les invariants,
— évite les effets cachés,
— rend le code plus robuste.

7 Données et fonctions

Les données décrivent un **état**. Les fonctions décrivent des **transformations**.

```
type Personne = {
    nom : string
    age : int
}

let estMajeur p = p.age >= 18
```

Les données sont passives, le comportement est porté par les fonctions.

8 Fonctions comme valeurs

Une fonction est une valeur à part entière.

```
let appliquerDeuxFois f x =
    f (f x)

appliquerDeuxFois carre 3   // 81
```

Ici, une fonction est passée en paramètre. C'est le principe des **fonctions d'ordre supérieur**.

9 Itération sans boucles

9.1 `map`

`map` applique une fonction à chaque élément d'une collection.

```
let nombres = [1; 2; 3; 4]
let carres = List.map carre nombres
```

Résultat :

```
[1; 4; 9; 16]
```

Le parcours est implicite. Seule la transformation est exprimée.

9.2 `filter`

`filter` sélectionne les éléments qui vérifient un prédicat.

```
let pairs = List.filter (fun x -> x % 2 = 0) nombres
```

Résultat :

```
[2; 4]
```

10 Le pipeline de fonctions (`|>`)

Le pipeline permet de lire le code **de gauche à droite**, comme une suite de transformations.

```
let resultat =
    nombres
    |> List.map carre
    |> List.filter (fun x -> x > 5)
```

Lecture :

- partir de `nombres`,
- calculer les carrés,
- garder ceux strictement supérieurs à 5.

Le pipeline améliore fortement la **lisibilité**.

11 `fold` : le modèle général

`fold` permet de réduire une collection à une seule valeur.

```
let somme =
    List.fold (fun acc x -> acc + x) 0 nombres
```

Ici :

- `acc` est l'accumulateur,
- `0` est la valeur initiale,
- l'état intermédiaire est **explicite**.

12 Récursion et récursion terminale

12.1 Récursion simple

```
let rec somme n =
    if n = 0 then 0
    else n + somme (n - 1)
```

Cette définition est logique, mais non terminale.

12.2 Récursion terminale

```
let somme n =
    let rec aux acc n =
        if n = 0 then acc
        else aux (acc + n) (n - 1)
    aux 0 n
```

L'accumulateur rend l'état explicite, ainsi, le compilateur peut optimiser.

13 Types : un outil de raisonnement

Les types servent à exprimer des **invariants**.

```
let longueur (s : string) : int =
    s.Length
```

La signature suffit à comprendre la fonction.

13.1 Types produits

13.1.1 Tuples

```
let point = (3, 4)
```

Type :

```
int * int
```

13.1.2 Records

```
type Point = { x : int; y : int }
```

Les records sont préférables pour le code métier.

13.2 Types sommes

```
type Direction =
    | Nord
    | Sud
    | Est
    | Ouest
```

Une valeur est exactement **un cas parmi plusieurs**.

13.3 `Option`

```
let chercher x liste =
    List.tryFind ((=) x) liste
```

Le résultat est de type `int option`.

13.4 `Result`

```
let diviser a b =
    if b = 0 then Error "division par zéro"
    else Ok (a / b)
```

L'erreur est visible dans le type.

14 Pattern matching

Le pattern matching permet de raisonner **par cas**, de façon exhaustive.

```
let decrireDirection d =
    match d with
    | Nord -> "haut"
    | Sud -> "bas"
    | Est -> "droite"
    | Ouest -> "gauche"
```

14.1 Déstructuration

```
let norme p =
    match p with
    | { x = x; y = y } -> sqrt (float (x*x + y*y))
```

La structure des données guide la logique.

15 Robustesse, lisibilité et complexité

Un code fonctionnel bien écrit est :

— lisible sans exécuter,
— robuste grâce aux types,
— explicite sur sa complexité (nombre de parcours).

Les optimisations viennent **après** la clarté.

16 Transfert vers Java moderne

Les mêmes idées existent en Java :

— lambdas,
— Streams,
— `map`, `filter`, `reduce`.

La syntaxe change, mais la **posture de raisonnement** reste valable.

17 Conclusion

La programmation fonctionnelle apporte :

— un raisonnement plus local,
— moins d'état caché,
— un code plus sûr et plus lisible.