



→ Digitalize society



# Formation F# 5.0

## *Les fonctions*



Décembre 2021

**SOAT.FR**

# About me



## Romain DENEAU

- SOAT depuis 2009
- Senior Developer C# F# TypeScript
- Passionné de Craft
- Auteur sur le blog de SOAT



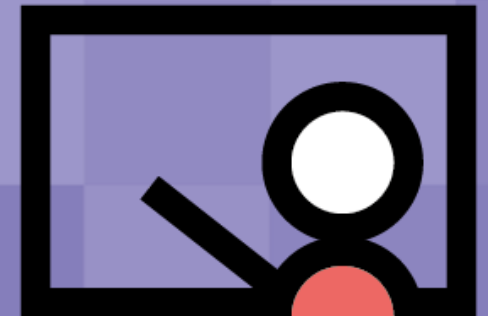
DeneauRomain



rdeneau

# Sommaire

- Signature des fonctions
- Fonctions
- Opérateurs
- Interop avec BCL .NET



# 1. Signature des fonctions



# Problèmes avec `void` en C#

`void` oblige à faire du spécifique = 2 fois + de boulot 😡

- 2 types de délégués : `Action` vs `Func<T>`
- 2 types de tâches : `Task` vs `Task<T>`

Exemple :

```
interface ITelemetry
{
    void Run(Action action);
    T Run<T>(Func<T> func);

    Task RunAsync(Func<Task> asyncAction);
    Task<T> RunAsync<T>(Func<Task<T>> asyncFunc);
}
```

C#

# De `void` à `Void`

- 👉 Le problème avec `void`, c'est que ce n'est ni un type, ni une valeur.
- 💡 Si on avait un type "`Void`", un *Singleton* du type :

```
public class Void
{
    public static readonly Void Instance = new Void();

    private Void() {}
}
```

C#

## De `void` à `Void` (2)

On peut définir les *helpers* suivants pour convertir vers `Void` :

```
public static class VoidExtensions
{
    // Action → Func<Void>
    public static Func<Void> AsFunc(this Action action)
    {
        action();
        return Void.Instance;
    }

    // Func<Task> → Func<Task<Void>>
    public async static Func<Task<Void>> AsAsyncFunc(this Func<Task> asyncAction)
    {
        await asyncAction();
        return Void.Instance;
    }
}
```

C#

# Simplification de **ITelemetry**

On peut écrire une implémentation par défaut (C# 8) pour 2 des 4 méthodes :

```
interface ITelemetry
{
    void Run(Action action) ⇒
        Run(action.AsFunc());

    T Run<T>(Func<T> func);

    Task RunAsync(Func<Task> asyncAction) ⇒
        RunAsync(asyncAction.AsAsyncFunc());

    Task<T> RunAsync<T>(Func<Task<T>> asyncFunc);
}
```

C#



# Void s'appelle Unit en F#

En F#, pas de fonction `void` mais des fonctions avec type de retour `Unit` / `unit`.

`unit` a une seule instance (d'où son nom), notée `()`

→ Utilisée en tant que dernière expression d'une fonction "void":

```
let voidFunction arg =  
    // ...  
    ()
```

F#

# Fonctions sans paramètre

`unit` sert aussi à modéliser des fonctions sans paramètre :

```
let oneParam arg = ...  
let noParam () = ... // ➡ Avec  
let noParam2() = ... // ➡ ou sans espace
```

F#

💡 Intérêt de la notation `()` : on dirait une fonction C#.

⚠ **Attention** : on a vite fait d'oublier les `()` !

- ➔ Oubli dans la déclaration → simple valeur plutôt que fonction
- ➔ Oubli dans l'appel → alias de la fonction sans l'exécuter

# Fonction `ignore`

En F#, tout est expression mais on peut insérer des expressions de type `unit`, par un exemple un `printf` avant de renvoyer la valeur

Problème : quand on appelle une fonction `save` pour enregistrer en base mais elle renvoie la valeur `true` ou `false` qu'on veut ignorer.

Solution : utiliser la fonction `ignore` de signature `'a → unit`.  
→ Qqsoit la valeur fournie en paramètre, elle l'ignore et renvoie `()`.

```
let save entity = true

let a =
    save "bonjour" // ⚠ Warning FS0020: Le résultat de cette expression a le type 'bool' et est implicitement ignoré.
    save "bonjour" ▷ ignore // 🍌
```

F#

# Signature d'une fonction en F#

Notation fléchée :

- Fonction à 0 paramètre : `unit → TResult`
- Fonction à 1 paramètre : `T → TResult`
- Fonction à 2 paramètres : `T1 → T2 → TResult`
- Fonction à 3 paramètres : `T1 → T2 → T3 → TResult`

**? Quiz** : Pourquoi plusieurs `→` plutôt que des `,` ? Cela indique quoi ?

# Curryfication

Syntaxe des fonctions F# : paramètres séparés par des espaces

→ Indique que les fonctions sont curryfiées

→ D'où les `→` dans la signature entre les paramètres

```
let fn () = result           // unit → TResult
let fn arg = ()              // T   → unit
let fn arg = result          // T   → TResult

let fn x y = (x, y)          // T1 → T2 → (T1 * T2)

// Equivalents, explicitement curryfiés :
let fn x = fun y → (x, y)    // 1. Avec une lambda
let fn x =                   // 2. Avec une sous fonction
    let fn' y = (x, y)       // N.B. `x` vient du scope
    fn'
```

F#

# Curryfication - Compilation .NET

👉 Fonction curryfiée compilée en méthode avec paramètres tuplés  
→ Vue comme méthode normale quand consommée en C#

Exemple : F# puis équivalent C# (version simplifiée de [SharpLab](#)):

```
module A =  
    let add x y = x + y  
    let value = 2 ▷ add 1
```

F#

```
public static class A  
{  
    public static int add(int x, int y) => x + y;  
    public static int value => 3;  
}
```

C#

# Conception unifiée des fonctions

Le type `unit` et la curryfication permettent de concevoir les fonctions simplement comme :

- **Prend un seul paramètre** de type quelconque
  - y compris `unit` pour une fonction "sans paramètre"
  - y compris une autre fonction (*callback*)
- **Renvoie une seule valeur** de type quelconque
  - y compris `unit` pour une fonction "ne renvoyant rien"
  - y compris une autre fonction

👉 **Signature universelle** d'une fonction en F# : `'T → 'U`

# Ordre des paramètres

Pas le même ordre entre C# et F#

- Dans méthode extension C#, l'objet `this` est le 1er paramètre
  - Ex : `items.Select(x ⇒ x)`
- En F#, "l'objet" est plutôt le **dernier paramètre** : style *data-last*
  - Ex : `List.map (fun x → x) items`

Style *data-last* favorise :

- Pipeline : `items ▷ List.map square ▷ List.sum`
- Application partielle : `let sortDesc = List.sortBy (fun i → -i)`
- Composition de fonctions appliquées partiellement jusqu'au param "data"

Formation F# 5.0 • `(List.map square) >> List.sum`



# Ordre des paramètres (2)

⚠ Friction avec BCL .NET car plutôt *data-first*

💡 Solution : wrapper dans une fonction avec params dans ordre sympa en F#

```
let startsWith (prefix: string) (value: string) =  
    value.StartsWith(prefix)
```

F#

💡 **Tips** : utiliser `Option.defaultValue` plutôt que `defaultArg` avec les options

- Fonctions font la même chose mais params `option` et `value` sont inversés
- `defaultArg option value` : param `option` en 1er 😞
- `Option.defaultValue value option` : param `option` en dernier 👍

# Ordre des paramètres (3)

De même, préférer mettre **en 1er** les paramètres les + statiques  
= Ceux susceptibles d'être prédéfinis par application partielle

Ex : "dépendances" qui seraient injectées dans un objet en C#

👉 Application partielle = moyen de simuler l'injection de dépendances

# 2. ■ Les fonctions



# Binding d'une fonction

```
let f x y = x + y + 1
```

- Binding réalisé avec mot clé `let`
- Associe à la fois un nom (`f`) et les paramètres (`x` et `y`)
- Annotation de type optionnelle pour paramètres et/ou retour
  - `let f (x: int) (y: int) : int = ...`
  - Sinon, inférence de type, avec possible généralisation auto
- Dernière expression → valeur de retour de la fonction
- Possible définition de sous-fonctions (non génériques)

# Fonction générique

- Dans beaucoup de cas, inférence marche avec généralisation auto
  - `let listOf x = [x] → (x: 'a) → 'a list`
- Annotation explicite de params génériques
  - `let f (x: 'a) = ...` (pas besoin de faire `f<'a>` grâce au `'` 👍)
- Annotation explicite avec inférence du type générique
  - `let f (list: list<_>) = ...`

# Fonction anonyme / (Expression) Lambda

Expression définissant une fonction

Syntaxe : `fun parameter1 parameter2 etc → expression`

👉 **À noter :**

- Mot clé `fun` obligatoire
- Flèche fine `→` (Java) ≠ flèche grasse `⇒` (C#, Js)

# Fonctions anonymes - Quelques usages

## 1. En argument d'une *high-order function*

- Pour éviter de devoir définir une fonction nommée
- Recommandée pour une fonction courte, pour que cela reste lisible

```
[1..10] ▷ List.map (fun i → i + 1) // ➡ () autour de la lambda
```

F#

```
// Versus en passant par une fonction nommée
```

```
let add1 i = i + 1
```

```
[1..10] ▷ List.map add1
```

⚠ Lambda inutile : `List.map (fun x → f x)`  $\equiv$  `List.map f`

## 2. Dans un *let binding* avec inférence

- Pour rendre explicite quand la fonction renvoie une fonction
- Sorte de curryfication manuelle
- À utiliser avec parcimonie

```
let add x y = x + y           // Version normale, curryfiée automatiquement
let add' x = fun y → x + y    // Idem avec une sous lambda
let add'' = fun x → (fun y → x + y) // Idem en totalement "lambda-isée"
```

F#



### 3. Dans un *let binding* annoté

- Signature de la fonction pré-définie sous forme d'un type
- Type "fonction" s'utilise un peu comme une `interface` C#
  - Pour contraindre implémentation à suivre signature
  - Ex : *Domain modelling made functional* par Scott Wlaschin

```
type Add = int → int → int
```

F#

# Fonction anonyme `function`

- Mot clé `function` permet aussi de définir une fonction anonyme
- Syntaxe abrégée équivalente à `fun x → match x with`
- Prend 1 paramètre qui est implicite

```
let ouiNon x =  
  match x with  
  | true  → "Oui"  
  | false → "Non"  
  
// Réécrit avec `function`  
let ouiNon = function  
  | true  → "Oui"  
  | false → "Non"
```

F#

👉 Pas de cas d'usage spécifique. Son emploi est une question de goût.

# Déconstruction de paramètres

- Comme en JavaScript, on peut déconstruire *inline* un paramètre
- C'est également une façon d'indiquer le type du paramètre
- Le paramètre apparaît sans nom dans la signature

Exemple avec un type *Record* 📌

```
type Person = { Name: string; Age: int }  
  
let name { Name = x } = x      // Person → string  
let age { Age = x } = x       // Person → int  
let age' person = person.Age  // Equivalent explicite  
  
let bob = { Name = "Bob"; Age = 18 } // Person  
let bobAge = age bob // int = 18
```

F#

# Déconstruction de paramètres (2)

On parle aussi de *pattern matching*

→ Mais je préfère réserver ce terme pour l'usage de `match x with ...`

Déconstruction pas adaptée pour un type union avec plusieurs cas 

→ Ex : liste F# (soit vide `[]`, soit valeur + sous-liste `head::tail`)

 **Solution** : faire un *pattern matching* de tous les cas de l'union

```
let printFirstItem (x::_) = // 'a list → unit
//          ~~~~ Warning FS0025: Critères spéciaux incomplets dans cette expression.
    printfn $"first element: {x}"

let printFirstItemOk = function
| x::_ → printfn $"first element: {x}"
| []   → printfn "none"
```

F#

# Paramètre tuple

- Comme en C#, on peut vouloir regrouper des paramètres d'une fonction
  - Par soucis de cohésion, quand ces paramètres forment un tout
  - Pour éviter le *code smell* [long\\_parameter\\_list](#)
- On peut les regrouper dans un tuple et même le déconstruire

```
// V1 : trop de paramètres
let f x y z = ...

// V2 : paramètres regroupés dans un tuple
let f params =
    let (x, y, z) = params
    ...

// V3 : idem avec tuple déconstruit sur place
let f (x, y, z) = ...
```

F#

# Paramètre tuple (2)

- `f (x, y, z)` ressemble furieusement à une méthode C# !
- La signature signale le changement : `(int * int * int) → TResult`
  - La fonction n'a effectivement plus qu'1! paramètre plutôt que 3
  - Perte possibilité application partielle de chaque élément du tuple

## 👉 Conclusion :

- Résister à la tentation d'utiliser tout le temps un tuple (*car familier* - C#)
- Réserver cet usage quand c'est pertinent de regrouper les paramètres
  - Sans pour autant déclarer un type spécifique pour ce groupe

# Fonction récursive

- Fonction qui s'appelle elle-même
- Syntaxe spéciale avec mot clé `rec` sinon erreur `FS0039: ... is not defined`
- Très courant en F# pour remplacer les boucles `for`
  - Car c'est souvent + facile à concevoir

Exemple : trouver nb étapes pour atteindre 1 dans la [suite de Syracuse](#) / Collatz

```
let rec steps (n: int) : int =  
    if n = 1 then 0  
    elif n % 2 = 0 then 1 + steps (n / 2)  
    else 1 + steps (3 * n + 1)
```

F#

# Tail recursion

- Type de récursivité où l'appel récursif est la dernière instruction
- Détecté par le compilateur et optimisé sous forme de boucle
  - Permet d'éviter les `StackOverflow`
- Procédé classique pouvant rendre tail récursif :
  - Ajouter un param "accumulateur", comme `fold` / `reduce`

```
let steps (number: int) : int =  
    let rec loop count n = // ➡ `loop` = nom idiomatique de ce type de fonction interne récursive  
        if n = 1 then count  
        elif n % 2 = 0 then loop (count + 1) (n / 2) // ➡ Dernier appel : `loop`  
        else loop (count + 1) (3 * n + 1) // ➡ idem
```

F#



# Fonctions mutuellement récursives

- Fonctions qui s'appellent l'une l'autre
- Doivent être déclarées ensemble :
  - 1ère fonction indiquée comme récursive avec `rec`
  - autres fonctions ajoutées à la déclaration avec `and`

```
// ⚠ Algo un peu alambiqué servant juste d'illustration
```

F#

```
let rec Even x =           // ➡ Mot clé `rec`  
    if x = 0 then true  
    else Odd (x-1)         // ➡ Appel à `Odd` définie + bas  
and Odd x =                // ➡ Mot clé `and`  
    if x = 0 then false  
    else Even (x-1)        // ➡ Appel à `Even` définie + haut
```

# Surcharge / *overload* de fonctions

⚠ Pas possible de surcharger une fonction

💡 Noms différents :

→ `List.map (mapping: 'T → 'U) list`

→ `List.mapi (mapping: (index: int) → 'T → 'U) list`

💡 Implémentation via fonction template 📝

# Fonction template

Permet de créer des "surcharges" spécialisées :

```
type ComparisonResult = Bigger | Smaller | Equal

// Fonction template, 'private' pour la "cacher"
let private compareTwoStrings (comparison: StringComparison) string1 string2 =
    let result = System.String.Compare(string1, string2, comparison)
    if result > 0 then
        Bigger
    else if result < 0 then
        Smaller
    else
        Equal

// Application partielle du paramètre 'comparison'
let compareCaseSensitive = compareTwoStrings StringComparison.CurrentCulture
let compareCaseInsensitive = compareTwoStrings StringComparison.CurrentCultureIgnoreCase
```

F#

# Fonction template (2)

👉 Emplacement du paramètre de spécialisation :

→ En C#, en dernier :

```
String.Compare(String, String, StringComparison)  
String.Compare(String, String)
```

C#




→ En F#, en premier pour permettre application partielle :

```
compareTwoStrings      : StringComparison → String → String → ComparisonResult  
compareCaseSensitive   :                  String → String → ComparisonResult
```

F#

# Organisation des fonctions

3 façons d'organiser les fonctions = 3 endroits où les déclarer :

- *Module* : fonction déclarée dans un module 
- *Nested* : fonction déclarée à l'intérieur d'une valeur / fonction
  -  Encapsuler des helpers utilisés juste localement
  -  Paramètres de la fonction chapeau accessibles à fonction *nested*
- *Method* : fonction définie comme méthode dans un type (*next slide*)

# Méthodes

- Définies avec mot-clé `member` plutôt que `let`
- Choix du *self-identifiant* : `this`, `me`, `_`...
- Paramètres sont au choix :
  - Tuplés : style OOP
  - Curryfiés : style FP

# Méthodes - Exemple

```
type Product = { SKU: string; Price: float } with // ➡ `with` nécessaire pour l'indentation
// Style avec tuplification et `this` // Alternative : `{ SKU...}` à la ligne
member this.TupleTotal(qty, discount) =
    (this.Price * float qty) - discount

// Style avec curryfication et `me`
member me.CurriedTotal qty discount = // ➡ `me` désigne le "this"
```

F#

# Fonction vs Méthode

Fonctionnalité	Fonction	Méthode
Nommage	camelCase	PascalCase
Curryfication	✓ oui	✓ si non tuplifiés
Paramètres nommés	✗ non	✓ si tuplifiés
Paramètres optionnels	✗ non	✓ si tuplifiés
Surcharge / <i>overload</i>	✗ non	✓ si tuplifiés
Inférence à l'usage	✓ oui	✗ non
En argument d'une <i>high-order fn</i>	✓ oui	✗ non, lambda nécessaire
Support du <code>inline</code>	✓ oui	✓ oui
Réursive	✓ si <code>rec</code>	✓ oui



# Fonctions standards

Définies dans `FSharp.Core` automatiquement importé

## Conversion

- `box`, `tryUnbox`, `unbox` : *boxing* et (tentative de) *unboxing*
- `byte`, `char`, `decimal`, `float`, `int`, `string` : conversion en `byte`, `char`, ...
- `enum<'TEnum>` : conversion en l'enum spécifié

# Math

- `abs`, `sign` : valeur absolue, signe (-1 si < 0...)
- `(a)cos(h)`, `(a)sin`, `(a)tan` : (co)sinus/tangente (inverse/hyperbolique)
- `ceil`, `floor`, `round` : arrondi (inf, sup)
- `exp`, `log`, `log10` : exponentielle, logarithme...
- `pown x (n: int)` : *power* = `x` à la puissance `n`
- `sqrt` : *square root* / racine carrée

# Autres

- `compare a b : int` : renvoie -1 si  $a < b$ , 0 si  $=$ , 1 si  $>$
- `hash` : calcul le hash (code)
- `max`, `min` : maximum et minimum de 2 valeurs comparables
- `ignore` : pour "avaler" une valeur et obtenir `unit`
- `id` : next slide 🙌

# Fonction `id`

Définition `let id x = x` • Signature : `(x: 'T) → 'T`

→ Fonction à un seul paramètre d'entrée

→ Qui ne fait que renvoyer ce paramètre

Pourquoi une telle fonction ?

→ Nom `id` = abréviation de `identity`

→ Zéro / Élément neutre de la composition des fonctions

Opération	Identité	Exemple
Addition <code>+</code>	<code>0</code>	<code>0 + 5 ≡ 5 + 0 ≡ 5</code>
Multiplication <code>*</code>	<code>1</code>	<code>1 * 5 ≡ 5 * 1 ≡ 5</code>
Composition <code>&gt;&gt;</code>	<code>id</code>	<code>id &gt;&gt; fn ≡ fn &gt;&gt; id ≡ fn</code>

# Fonction `id` - Cas d'utilisation

Avec une *high-order function* faisant 2 choses :

- 1 opération
- 1 mapping de valeur via param `'T → 'U`

Ex : `List.collect fn list` = flatten + mapping

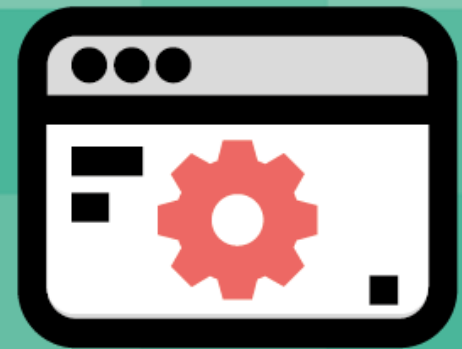
Comment faire juste l'opération et pas de mapping ?

→ `list ▷ List.collect (fun x → x)` 🙄

→ `list ▷ List.collect id` 👍

→ 🙌 Meilleure alternative : `List.concat list` 100

# 3 ■ Les opérateurs



# Opérateur

Est défini comme une fonction

- Opérateur unaire : `let (~symbols) = ...`
- Opérateur binaire : `let (symbols) = ...`
- *Symbols* = combinaison de `% & * + - . / < = > ? @ ^ | ! $`

2 façons d'utiliser les opérateurs

- En tant qu'opérateur → infixe `1 + 2` ou préfixe `-1`
- En tant que fonction → chars entre `()` : `(+) 1 2`  $\equiv$  `1 + 2`

# Opérateurs standards




Également définis dans `FSharp.Core`

- Opérateurs arithmétiques : `+`, `-` ...
- Opérateurs de pipeline
- Opérateurs de composition



# Opérateurs *Pipe*

Opérateurs binaires, placés entre une valeur simple et une fonction

- Appliquent la valeur à la fonction = Passe la valeur en argument
- Permettent d'éviter la mise entre parenthèses / précedence
- ∃ plusieurs *pipes*
  - *Pipe right*  : le *pipe* "classique"
  - *Pipe left*  a.k.a. *pipe* inversé
  - *Pipe right 2* 
  - Etc.

# Opérateur *Pipe right*

Inverse l'ordre entre fonction et valeur : `val ▷ fn`  $\equiv$  `fn val`

- Ordre naturel "sujet verbe", comme appel méthode d'un objet (`obj.M(x)`)
- *Pipeline* : enchaîner appels de fonctions, sans variable intermédiaire
- Aide inférence d'objet. Exemple :

```
let items = ["a"; "bb"; "ccc"]  
  
let longestKo = List.maxBy (fun x → x.Length) items // ✗ Error FS0072  
// ~~~~~  
  
let longest = items ▷ List.maxBy (fun x → x.Length) // ✔ Renvoie "ccc"
```

F#

# Opérateur *Pipe left* ◁

`fn ◁ expression`  $\equiv$  `fn (expression)`

- 🙅 Usage un peu moins courant que ▶
- ✅ Avantage mineur : permet d'éviter des parenthèses
- ❌ Inconvénient majeur : se lit de droite à gauche  
→ Inverse du sens lecture naturel en anglais et ordre exécution

```
printf "%i" 1+2           // ✨ Erreur  
printf "%i" (1+2)         // Avec parenthèses  
printf "%i" ◁ 1+2         // Avec pipe inversé
```

F#

## Opérateur *Pipe left* (2)

Quid d'une expression telle que `x ▷ fn ◁ y` ?

Exécutée de gauche à droite :

`(x ▷ fn) ◁ y`  $\equiv$  `(fn x) ◁ y`  $\equiv$  `fn x y`

- En théorie : permettrait d'utiliser `fn` en position infixée
- En pratique : difficile à lire à cause du double sens de lecture !

👉 Conseil : **À ÉVITER**

# Opérateur *Pipe right* 2

`(x, y) ▷ fn ≡ fn x y`

- Pour passer 2 arguments à la fois, sous la forme d'un tuple
- Usage peu fréquent, par exemple avec `fold` pour passer liste & seed

```
let items = [1..5]

// 😞 On peut manquer le 0 au bout (le seed)
let sumOfEvens = items ▷ List.fold (fun acc x → if x % 2 = 0 then acc + x else acc) 0

let sumOfEvens' =
    (0, items)
    ▷ List.fold (fun acc x → if x % 2 = 0 then acc + x else acc)

// 💡 Remplacer lambda par fonction nommée
let addIfEven acc x = if x % 2 = 0 then acc + x else acc
let sumOfEvens'' = items ▷ List.fold addIfEven 0
```

F#

# Opérateur *Compose* >>

Opérateurs binaires, placés **entre deux fonctions**

→ Le résultat de la 1ère fonction servira d'argument à la 2e fonction

`f >> g`  $\equiv$  `fun x → g (f x)`  $\equiv$  `fun x → x ▷ f ▷ g`

💡 Peut se lire « `f` ensuite `g` »

⚠ Les types doivent correspondre : `f: 'T → 'U` et `g: 'U → 'V`

→ On obtient une fonction de signature `'T → 'V`

```
let add1 x = x + 1
let times2 x = x * 2
```

```
let add1Times2 x = times2(add1 x) // 😞 Style explicite mais + chargé
let add1Times2' = add1 >> times2 // 👍 Style concis
```

F#

# Opérateur *Compose inverse* <<

Sert rarement, sauf pour retrouver un ordre naturel des termes

Exemple avec opérateur `not` (qui remplace le `!` du C#) :

```
let Even x = x % 2 = 0

// Pipeline classique
let Odd x = x ▷ Even ▷ not

// Réécrit avec composition inverse
let Odd = not << Even
```

F#

# *Pipe* ▶ ou *Compose* >> ?

**Compose** `let h = f >> g`

- Composition de 2 fonctions `f` et `g`
- Renvoie une nouvelle fonction
- Les fonctions `f` et `g` ne sont exécutées que lorsque `h` l'est

**Pipe** `let result = value ▶ f`

- Juste une syntaxe différente pour passer un argument
- La fonction `f` est :
  - Exécutée si elle n'a qu'1! param → `result` est une valeur
  - Appliquée partiellement sinon → `result` est une fonction



# Style *Point-free*

A.k.a *Programmation tacite*

Fonction définie par composition ou application partielle ou avec **function**  
→ **Paramètre implicite**, d'où le « sans-point » (dans l'espace)

```
let add1 x = x + 1           // (x: int) → int
let times2 x = x * 2        // (x: int) → int
let add1Times2 = add1 >> times2 // int → int • x implicite • Par composition

let isEven x = x % 2 = 0
let evens list = List.filter isEven list // (list: int list) → int list
let evens' = List.filter isEven // int list → int list • Par application partielle

let greet name age = printfn $"My name is {name} and I am %d{age} years old!" // name:string → age:int → unit
```

F#

# Style *Point-free* - Pros/Cons

## ✓ Avantages

Style concis • Abstraction des paramètres, opère au niveau fonctions

## ✗ Inconvénients

Perd le nom du paramètre devenu implicite dans la signature

→ Sans importance si la fonction reste compréhensible :

- Nom du param non significatif (ex. `x`)
- Type du param et nom de la fonction suffisent

→ Déconseillé pour une API publique

# Style *Point-free* - Limite

Marche mal avec fonctions génériques :

```
let isEmptyKo = not << List.isEmpty // ✨ Error FS0030: Restriction de valeur
let isEmpty<'a> = not << List.isEmpty<'a> // ⚠ Avec annotation
let isEmpty' list = not (List.isEmpty list) // ⚠ Style explicite
```

F#

<https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/conventions#partial-application-and-point-free-programming>

# Fonction **inline** : principe

 [https://fr.wikipedia.org/wiki/Extension\\_inline](https://fr.wikipedia.org/wiki/Extension_inline)

“ **Extension inline** ou **inlining** : optimisation d'un compilateur qui remplace un appel de fonction par le code (*le corps*) de cette fonction. ”

→ Gain de performance 👍

→ Compilation + longue ⚠️

💡 Même principe que les refactoros *Inline Method*, *Inline Variable*

# Fonction `inline` (2)

Mot clé `inline` indique au compilateur de "*inliner*" la fonction  
→ Usage typique : petite fonction/opérateur de "sucre syntaxique"

```
// See https://github.com/dotnet/fsharp/blob/main/src/fsharp/FSharp.Core/prim-types.fs  
let inline (▷) x f = f x  
let inline ignore _ = ()  
  
let t = true ▷ ignore  
    ~= ignore true // Après inline du pipe  
    ~= ()           // Après inline de ignore
```

F#

# Opérateurs personnalisés

2 possibilités :

- Surcharge d'opérateurs
- Création d'un nouvel opérateur

# Surcharge d'opérateurs

En général, concerne un type spécifique

→ Surcharge définie à l'intérieur du type associé (*comme en C#*)

```
type Vector = { X: int; Y: int } with
    // Opérateur unaire (cf ~ et ! param) d'inversion d'un vecteur
    static member (~) (v: Vector) =
        { X = -v.X
          Y = -v.Y }

    // Opérateur binaire d'addition de 2 vecteurs
    static member (+) (a: Vector, b: Vector) =
        { X = a.X + b.X
          Y = a.Y + b.Y }

let v1 = ~{ X=1; Y=1 } // { X = -1; Y = -1 }
let v2 = { X=1; Y=1 } + { X=1; Y=3 } // { X = 2; Y = 4 }
```

F#

# Création d'un nouvel opérateur

- Définition plutôt dans un module ou dans un type associé
- Cas d'usage classique : alias fonction existante, utilisé en infixe

```
// "OR" Composition of 2 functions (fa, fb) which return an optional result
let (<||>) fa fb x =
    match fa x with
    | Some v → Some v // Return value produced by (fa x) call
    | None   → fb x   // Return value produced by (fb x) call

// Functions: int → string option
let tryMatchPositiveEven x = if x > 0 && x % 2 = 0 then Some $"Even {x}" else None
let tryMatchPositiveOdd  x = if x > 0 && x % 2 <> 0 then Some $"Odd {x}"  else None
let tryMatch = tryMatchPositiveEven <||> tryMatchPositiveOdd

tryMatch 0;; // None
tryMatch 1;; // Some "Odd 1"
tryMatch 2;; // Some "Even 2"
```

F#



# Symboles autorisés dans un opérateur

## Opérateur unaire "tilde"

→ `~` suivi de `+`, `-`, `+`, `-`, `%`, `%`, `&`, `&&`

## Opérateur unaire "snake"

→ Plusieurs `~`, ex : `~~~~`

## Opérateur unaire "bang"

→ `!` suivi combinaison de `!`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `=`, `>`, `@`, `^`, `|`, `~`, `?`

→ Sauf `≠` (!=) qui est binaire

## Opérateur binaire

→ Toute combinaison de `!`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `=`, `>`, `@`, `^`, `|`, `~`, `?`

→ qui ne correspond pas à un opérateur unaire

# Symboles à l'usage

Tout opérateur s'utilise tel quel

! Sauf opérateur unaire "tilde" : s'utilise sans le `~` initial

Opérateur	Déclaration	Usage
Unaire tilde	<code>let (~&amp;&amp;) x = ...</code>	<code>&amp;&amp;x</code>
Unaire snake	<code>let (~~~) x = ...</code>	<code>~~~x</code>
Unaire bang	<code>let (!!!) x = ...</code>	<code>!!!x</code>
Binaire	<code>let (&lt;^&gt;) x y = ...</code>	<code>x &lt;^&gt; y</code>

👉 Espace obligatoire entre `(` et `*` pour distinguer d'un commentaire `(* *)`

→ `let ( *+ ) x y = x * y + y` ✓

# Opérateur ou fonction ?

## Opérateur infixe vs fonction

👍 **Pour** : ordre naturel de lecture (gauche → droite) • éviter parenthèses  
→ `1 + 2 * 3` vs `multiply (add 1 2) 3`

⚠️ **Contre** : opérateur folklorique (ex : `@!`) moins compréhensible que fonction dont le nom utilise le **langage du domaine**

## Utiliser opérateur en tant que fonction

👍 Application partielle : `(+) 1`  $\equiv$  `fun x → x + 1` • `let isPositive = (<) 0`

💡 Composition : `(~) >> ((* ) 2)`  $\equiv$  `fun x → -x * 2`

# Pipeline avec méthode d'instance

Exemple : appeler `ToLower()` d'une `string`

- Via lambda : `"MyString" ▷ (fun x → x.ToLower())`
- Idem via fonction nommée telle que :
  - `String.ToLower` de la librairie [FSharpPlus](#)
  - `"MyString" ▷ String.ToLower`
- Pipeline Valeur intermédiaire : `let low = "MyString".ToLower()`

# Mémoïsation

💡 **Idée** : réduire le temps de calcul d'une fonction

? **Comment** : mise en cache des résultats

→ Au prochain appel avec mêmes arguments, renverra résultat en cache

👉 **En pratique** : fonction `memoizeN` de la librairie [FSharpPlus](#)

⚠️ **Attention** : Comme toute optimisation, à utiliser quand le besoin se fait sentir et en validant (mesurant) que cela marche sans désagrément annexe.

👉 Ne pas confondre avec expression `lazy` (*slide suivante*)

# Lazy expression

Sucre syntaxique pour créer un objet .NET `Lazy<'T>` à partir d'une expression

- Expression pas évaluée immédiatement mais qu'à la 1ère demande (*Thunk*)
- Intéressant pour améliorer performance sans trop complexifier le code

```
let printAndForward x = printfn $"{x}"; x

let a = lazy (printAndForward "a")

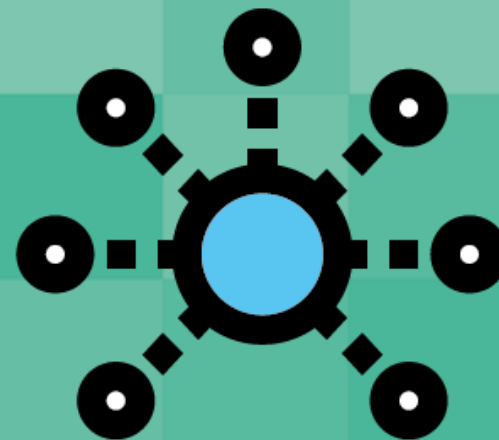
let b = printAndForward "b"
// > b

printfn $"{a.Value} et {b}"
// > a
// > a et b

printfn $"{a.Value} et c"
// > a et c
```

F#

# 4 ■ Interop avec BCL



BCL = Base Class Library .NET

# Appel à une méthode de la BCL

Méthode est "pseudo-tuplifiée"

- Tous les arguments doivent être spécifiés (1)
- Application partielle des paramètres non supportée (2)
- Mais ne marche pas avec un vrai tuple F# (3)

```
System.String.Compare("a", "b") // ✓ (1)
System.String.Compare ("a","b") // ✓

System.String.Compare "a" "b"    // ✗ (2)
System.String.Compare "a","b"    // ✗

let tuple = ("a","b")
System.String.Compare tuple      // ✗ (3)
```

F#



# Paramètre **out** - En C#

**out** utilisé pour avoir plusieurs valeurs en sortie

→ Ex : `Int32.TryParse`, `Dictionary<,>.TryGetValue` :

```
if (int.TryParse(maybeInt, out var value))  
    Console.WriteLine($"It's the number {value}.");  
else  
    Console.WriteLine($"{maybeInt} is not a number.");
```

C#

# Paramètre **out** - En F#

Possibilité de consommer la sortie sous forme de tuple 👍

```
match System.Int32.TryParse maybeInt with  
| true, i  → printf $"It's the number {value}."  
| false, _ → printf $"{{maybeInt}} is not a number."
```

F#

💡 Fonctions F# **tryXxx** s'appuient plutôt sur le type **Option<T>** 📌

# Instancier une classe avec `new` ?

```
// (1) `new` autorisé mais non recommandé
type MyClass(i) = class end

let c1 = MyClass(12)      // 👍
let c2 = new MyClass(234) // 👎 mais pas idiomatique

// (2) IDisposable ⇒ `new` obligatoire et `use` plutôt que `let`
open System.IO
let fn () =
    let _ = FileStream("hello.txt", FileMode.Open)
    // ⚠ Warning : Il est recommandé que les objets prenant en charge
    // l'interface IDisposable soient créés avec la syntaxe 'new Type(args)'

    use f = new FileStream("hello.txt", FileMode.Open)
    f.Close()
```

F#

# Appel d'une méthode surchargée

- Compilateur peut ne pas comprendre quelle surcharge est appelée
- Astuce : faire appel avec argument nommé

```
let createReader fileName =  
    new System.IO.StreamReader(path=fileName)  
    // 🙌 Param `path` → `filename` inféré en `string`  
  
let createReaderByStream stream =  
    new System.IO.StreamReader(stream=stream)  
    // 🙌 Param `stream` de type `System.IO.Stream`
```

F#

# 5. Quiz



# Question 1

Comment définir la valeur de retour `v` d'une fonction `f` ?

A. Il suffit de nommer la valeur `result`

B. Faire `return v`

C. `v` constitue la dernière ligne de `f`

🕒 10''



# Réponse 1

Comment définir la valeur de retour `v` d'une fonction `f` ?

A. Il suffit de nommer la valeur `result` ❌

B. Faire `return v` ❌

C. `v` constitue la dernière ligne de `f` ✅



## Question 2

Comment écrire fonction **add** prenant 2 **string**s et renvoyant un **int**

- A. `let add a b = a + b`
- B. `let add (a: string) (b: string) = (int a) + (int b)`
- C. `let add (a: string) (b: string) : int = a + b`

🕒 20"





# Réponse 2

Comment écrire fonction `add` prenant 2 `string`s et renvoyant un `int`

A. `let add a b = a + b` ❌

| Mauvais type inféré pour `a` et `b` : `int`

B. `let add (a: string) (b: string) = (int a) + (int b)` ✅

| Il faut spécifier le type de `a` et `b`.

| Il faut les convertir en `int`.

| Le type de retour `int` peut être inféré.

C. `let add (a: string) (b: string) : int = a + b`

| Ici, `+` concatène les `string`s.



## Question 3

Que fait le code `add >> multiply` ?

- A. Créer un pipeline
- B. Définir une fonction
- C. Créer une composition

 10''



# Réponse 3

Que fait le code `add >> multiply` ?

- A. Créer un pipeline ❌
- B. Définir une fonction ❌
- C. Créer une composition ✅



## Question 4

Retrouvez le nom de ces fonctions Core

A. `let ? _ = ()`

B. `let ? x = x`

C. `let ? f x = f x`

D. `let ? x f = f x`

E. `let ? f g x = g (f x)`

🕒 60" 💡 Il peut s'agir d'opérateurs



# Réponse 4

- A. `let inline ignore _ = ()`  
→ **Ignore** : [prim-types.fs#L459](#)
- B. `let id x = x`  
→ **Identity** : [prim-types.fs#L4831](#)
- C. `let inline (<) func arg = func arg`  
→ **Pipe Left** : [prim-types.fs#L3914](#)
- D. `let inline (>) arg func = func arg`  
→ **Pipe Right** : [prim-types.fs#L3908](#)
- E. `let inline (>>) func1 func2 x = func2 (func1 x)`  
→ **Compose Right** : [prim-types.fs#L3920](#)



## Question 5. Que signifie ces signatures ?

Combien de paramètres ? De quel type ? Type du retour ?

A. `int → unit`

B. `unit → int`

C. `string → string → string`

D. `('T → bool) → 'T list → 'T list`

 60''



## Réponse 5. Que signifie ces signatures ?

A. `int → unit`

1 paramètre `int` - pas de valeur renvoyée

B. `unit → int`

aucun paramètre - renvoie un `int`

C. `string → string → string`

2 paramètres `string` - renvoie une `string`

D. `('T → bool) → 'T list → 'T list`

2 paramètres : un prédicat et une liste - renvoie une liste  
→ Fonction `filter`



## Question 6. Signature de `h` ?

```
let f x = x + 1
let g x y = $"%i{x} + %i{y}"
let h = f >> g
```

F#

- A. `int → int`
- B. `int → string`
- C. `int → int → string`
- D. `int → int → int`

🕒 30" 💡 `%i{a}` indique que `a` est un `int`





## Réponse 6. Signature de **h** ?

C. `int → int → string` ✓

```
let f x = x + 1 → f: (x: int) → int  
» 1 → int → x: int → x + 1: int
```

```
let g x y = $"{+x} + {+y}" → (x: int) → (y: int) → string  
» %i{x} → int  
» $" ... " → string
```

```
let h = f >> g  
» h peut s'écrire let h x y = g (f x) y  
» Même x que f → int, même y que g → int
```



## Annexe 6. Signature de `h` ?

```
let f x = x + 1
let g x y = $"%i{x} + %i{y}"
let h = f >> g
```

F#

👉 **Conseil** : éviter d'utiliser `>>` avec des fonctions d'arité différente (*ici 1 pour `f`, 2 pour `g`*) car ce n'est pas lisible



## Question 7. Combien vaut `f 2` ?

```
let f = (-) 1;  
f 2 // ?
```

F#

A. 1

B. 3

C. -1

🕒 10''



## Réponse 7. Combien vaut `f 2` ?

```
let f = (-) 1  
f 2 // ?
```

F#

C. `-1`

Contre-intuitif : on s'attend à ce que `f` décrémente de 1.

On comprend en écrivant `f` ainsi :

```
let f x = 1 - x
```



La fonction qui décrémente de 1 peut s'écrire :

```
let f = (+) -1 ou let f x = x - 1
```



# 6 ■ Le Récap'



Ça en fait des choses juste sur les fonctions 😊

- Signature avec notation fléchée
- Signature universelle `T → U` grâce à type `Unit` et curryfication
- Fonction générique, fonction anonyme/lambda
- Fonction récursive et *tail recursion*
- Différences entre fonctions et méthodes
- Fonctions et opérateurs standards, dont `>`, `>>`
- Surcharger ou créer opérateur
- Notation *point-free*
- Interopérabilité avec la BCL



Merci 🙏

**SOAT**

→ Digitalize society



**SOAT.FR**