



→ Digitalize society



Formation F# 5.0

Bases



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



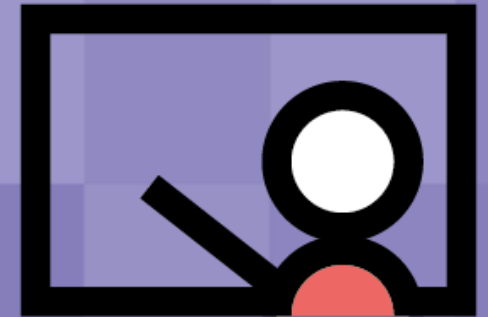
DeneauRomain



rdeneau

Sommaire

- Le F#, c'est quoi ?
- Syntaxe : fondamentaux, indentation
- Premiers concepts
 - Currification et application partielle
 - Tout est expression
 - Inférence de type



👉 Notes préalables

1. Symbole 📌 : indique notion abordée + tard
2. Code utilise la police **Fira Code** - [🔗 github.com/tonsky/FiraCode](https://github.com/tonsky/FiraCode) :

```
'→' = '-' + '>'      '≤' = '<' + '='      '=' = '=' + '='  
'⇒' = '=' + '>'      '≥' = '>' + '='      '≠' = '!' + '='  
'◇' = '<' + '>'      '▷' = '|' + '>'      '◁' = '<' + '|'  
'□' = '[' + '|'      '□' = '|' + ']'
```

JS

- 💡 Setting dans VsCode pour activer la ligature : `"editor.fontLigatures": true`
- 💡 Dans Rider, idem avec police **JetBrains Mono** - [🔗 jetbrains.com/lp/mono/](https://jetbrains.com/lp/mono/)

1. ■ Le F#, c'est quoi ?



Points clés

Famille des langages Microsoft - Plateforme **.NET**

- Son concepteur : Don Syme @ Microsoft Research
- \simeq Implémentation de OCaml pour .NET
- \simeq Inspirée par Haskell (*Version 1.0 en 1990*)
- `dotnet new -lang F#`
- Inter-opérabilité entre projets/assemblies C# et F#

Langage multi-paradigme ***Functional-first*** et très concis

Là où C# est *imperative/object-oriented-first* et plutôt verbeux
(même s'il s'inspire de F# pour être + succinct)

Historique

Date	C#	F#	.NET	Visual Studio
2002	C# 1.0		.NET Framework 1.0	VS .NET 2002
2005		F# 1.x	.NET Framework 1.0	VS 2005 ?
2010	C# 4.0	F# 2.0	.NET Framework 4	VS 2010
2015	C# 6.0	F# 4.0	.NET Framework 4.6, .NET Core 1.x	VS 2015
2018	C# 7.3	F# 4.5	.NET Framework 4.8, .NET Core 2.x	VS 2017
2019	C# 8.0	F# 4.7	.NET Core 3.x	VS 2019
2020	C# 9.0	F# 5.0	.NET 5.0	VS 2019

Éditeurs / IDE

VsCode + [Ionide](#)

→ 🙅 Permissif : ne remonte pas toujours toutes les erreurs de compilation

Visual Studio / Rider

→ 🙅 Moins de refacto que pour C#

<https://try.fsharp.org/>

→ Online [REPL](#) avec exemples

Rappel : setup du poste

<https://docs.microsoft.com/en-us/learn/modules/fsharp-first-steps/4-set-up-development-environment-exercise>

- Installation du SDK .NET (5.0 min, 6.0 si dispo)
- Installation de VScode
- Ajout de l'extension Ionide-fsharp

(Optionnel) extensions complémentaires : <https://www.compositional-it.com/news-blog/fantastic-f-and-azure-developer-extensions-for-vscode/>

F# interactive (FSI)

- REPL disponible dans VS, Rider, vscode + `dotnet fsi`
- Usage : vérifier en live un bout de code
 - 💡 Terminer expression par `::` pour l'évaluer
- Existe depuis le départ (*cf. aspect scripting du F#*)
 - C# interactive + récent (VS 2015 Update 1)
- Alternative : [LINQPad](#)



Démo

Types de fichier

4 types de fichier : `.fs`, `.fsi`, `.fsx`, `.fsproj`

- Mono langage : purement pour/en F#
- Standalone vs Projet

Fichier standalone

- Fichier de script `.fsx`
 - Exécutable (*d'où le x*) dans la console FSI
 - Indépendant mais peut référencer autre fichier, DLL, package NuGet.

Fichiers de projet

- En C# : `.sln` contient `.csproj` qui contient `.cs`
- En F# : `.sln` contient `.fsproj` qui contient `.fs` et `.fsi`
 - Fichier projet `.fsproj`
 - Fichier de code `.fs`
 - Fichier de signature `.fsi` (*i* comme *interface*)
 - Associé à un fichier `.fs` de même nom
 - Optionnel et plutôt rare -- + d'info : [MSDN](#)
 - Renforcer encapsulation (*idem* `.h` en C)
 - Séparer longue documentation (xml-doc)

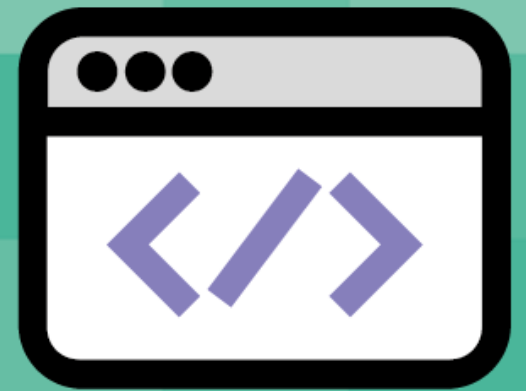
💡 **Interop C# - F#** = Mixer `.csproj` et `.fsproj` dans `.sln` ... 📌

Projet F#

Création dans un IDE ou avec la CLI `dotnet` :

- `dotnet new -l` : lister les types de projet supportés
- `dotnet new console --language F# -o MyFSharpApp`
- Création d'un projet console nommé `MyFSharpApp`
- `--language F#` à spécifier ; sinon C#
- `dotnet build` : builder le projet
- `dotnet run` : builder le projet et lancer l'exécutable résultant

2 ■ Syntaxe - Fondamentaux



Syntaxe - Clé

1er point fort de F# : langage succinct

Pour sans rendre compte :

1. Passons rapidement en revue sa syntaxe
2. Ensuite vous pourrez commencer à jouer avec
 - 🙌 C'est à l'usage que l'on mesure le côté succinct de F#

Commentaires

```
(* This is block  
   comment *)
```

```
// And this is line comment
```

```
/// XML doc summary
```

```
/// <summary>  
/// Full XML doc  
/// </summary>
```

F#

Variables Valeurs


- Mot clé `let` pour déclarer/nommer une valeur
- Pas besoin de `;` en fin de déclaration
- Liaison/*Binding* est immuable par défaut
 - \simeq `const` en JS, `readonly` pour un membre en C#
- Mutable avec `let mutable` et opérateur d'assignation `←`
 - \simeq `let` en JS, `var` en C#
 - Avec parcimonie, sur *scope* limité

```
let x = 1
x ← 2 // ✗ Error FS0027: Cette valeur n'est pas mutable.

let mutable x = 1
x ← 2 // ✔ Autorisé
```

F#

Noms

- Mêmes contraintes qu'en C#
- Sauf apostrophe ' 
 - permise dans nom au milieu ou à la fin (*mais pas au début*)
 - en fin de nom → indique une variante (*convention*)
- Entre doubles *backticks* → acceptent tout char (sauf saut de ligne)

```
let x = 1
let x' = x + 1 // Se prononce "x prime" ou "x tick"

let if' b t f = if b then t else f

let ``123 456`` = "123 456"
// 💡 Auto-complétion : pas besoin de taper les `` , directement 123 (quand ça veut marcher)
```

F#

Shadowing

- Consiste à redéfinir une valeur avec un nom existant
- En F#, interdit dans un même *scope* mais autorisé dans un sous-scope
 - Mais pas recommandé, sauf cas particulier

```
let a = 2

let a = "ko" // ✨ Error FS0037: Définition dupliquée de value 'a'

let b =
    let a = "ok" // 🙌 Pas d'erreur de compilation
    // `a` vaut "ok" (et pas 2) dans tout le reste de `b`
    // ...
```

F#

Annotation de type

- Optionnelle grâce à l'inférence
- Type déclaré après nom `name: type` (*comme en TypeScript*)
- Valeur obligatoire, même si `mutable`

```
let x = 1          // Inféré (int)
let y: int = 2     // Explicite

let z1: int        // ✨ Error FS0010: Construction structurée incomplète à cet emplacement...
```

F#

Constante

- *What*: Variable effacée à la compilation, remplacée par sa valeur
 - \simeq `const` C# - même idée en TS que `const enum`
- *How*: Valeur décorée avec attribut `Literal`
- Convention de nommage : PascalCase

```
[<Literal>] // Saut de ligne nécessaire car avant le `let`  
let AgeOfMajority = 18  
  
let [<Literal>] Pi = 3.14 // Possible aussi après le `let`
```

F#

Nombre

```
let pi = 3.14           // val pi : float = 3.14           • System.Double
let age = 18            // val age : int = 18               • System.Int32
let price = 5.95m       // val price : decimal = 5.95M     • System.Decimal
```

F#

⚠ Pas de conversion implicite entre nombre

→ 💡 Utiliser fonctions `int`, `float`, `decimal`

```
float 3;;              // val it : float = 3.0
decimal 3;;            // val it : decimal = 3M
int 3.6;;              // val it : int = 3
int "2";;              // val it : int = 2
```

F#

String

```
let name = "Bob"           // val name : string = "Bob"
let name2 = $"{name} Marley" // val name' : string = "Bob Marley"
let initial = name2.[0]     // val initial : char = 'B'
let firstName = name2.[0..2] // val firstName : string = "Bob"
```

F#

- `${val}` chaîne avec interpolation (⚠ F# 5)
- Avant F# 5 → utiliser `sprintf`
- `.[0]` : accès par index → caractère
- `.[0..2]` : accès par plage → sous-chaîne
- Alternative à méthode `Substring(index [, length])`

String (2)

```
// Verbatim string: idem C#  
let verbatimXml = @"<book title=""Paradise Lost"">"  
  
// Triple-quoted string : pas besoin d'échapper les guillemets `"`  
let tripleXml = """<book title="Paradise Lost">"""  
  
// Backslash strings : trim les espaces à gauche  
let poem =  
    "The lesser world was daubed\n\  
    By a colorist of modest skill\n\  
    A master limned you in the finest inks\n\  
    And with a fresh-cut quill."
```

F#

Listes

Liste immuable → type spécial F# ≠ `System.Collection.Generic.List<T>`

```
let abc = [ 'a'; 'b'; 'c' ] // val abc : char list = ['a'; 'b'; 'c']  
let a =  
    [ 2  
      3 ] // val a : int list = [2; 3]
```

F#

- Création avec `[]`, éléments séparés par `;` ou saut de ligne + indentation
 - ⚠ Piège : ne pas utiliser `,` sinon on a 1! élément : un tuple ⚠
- Notation ML du type `int list` = `List<int>`
 - 👉 Idiomatique que pour `list` et `option` ⚠

Listes - Opérateurs

```
let nums = [2..5]           // val nums : int list = [2; 3; 4; 5]
let nums' = 1 :: nums        // val nums' : int list = [1; 2; 3; 4; 5]

let chars = [ 'a' .. 'd' ]   // val chars : char list = ['a'; 'b'; 'c'; 'd']
let chars' = chars @ [ 'e'; 'f' ] // val chars' : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']
let c = chars.[0]            // val c : char = 'a'
```


- `min .. max` plage d'entiers entre min et max inclus
- `min .. step .. max` pour un écart > 1
- `::` opérateur *Cons* (signifiant construction)
- Ajoute un élément en tête de liste
- `@` opérateur *Append* = Concatène 2 listes
- Point `.` nécessaire pour accès par index `.[index]`

Listes - Module **List**

F# List	C# LINQ	JS Array
<code>map</code> , <code>collect</code>	<code>Select()</code> , <code>SelectMany()</code>	<code>map()</code> , <code>flatMap()</code>
<code>exists</code> , <code>forall</code>	<code>Any(predicate)</code> , <code>All()</code>	<code>some()</code> , <code>every()</code>
<code>filter</code>	<code>Where()</code>	<code>filter()</code>
<code>find</code> , <code>tryFind</code>	×	<code>find()</code>
<code>fold</code> , <code>reduce</code>	<code>Aggregate([seed])</code>	<code>reduce()</code>
<code>average</code> , <code>sum</code>	<code>Average()</code> , <code>Sum()</code>	×

👉 Autres fonctions : cf. documentation
<https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html>

Fonctions

- Fonction nommée : déclarée avec `let`
- Convention de nommage : **camelCase**
- Pas de `return` : renvoie toujours dernière expression
- Pas de `()` autour de tous les paramètres
 - `()` autour d'un paramètre avec type explicite ou déconstruit 

```
let square x = x * x // Fonction à 1 paramètre
let res = square 2    // Vaut 4
```

```
// Avec types explicites - juste pour l'exemple - pas idiomatique !
let square' (x: int) : int = x * x
```

F#

Fonctions de 0-n paramètres

- Paramètres et arguments séparés par **espace**
 - ⚠ Piège : `,` sert à instancier 1 tuple `! = 1!` paramètre
- `()` : fonction sans paramètre, sans argument
- Sans `()`, on déclare une valeur "vide", pas une fonction :

```
let add x y = x + y // Fonction à 2 paramètres
let res = add 1 2   // `res` vaut 3

let printHello () = printfn "Hello" // Fonction sans paramètre
printHello ()                     // Affiche "Hello" en console

let notAFunction = printfn "Hello" // Affiche "Hello" en console et renvoie "vide"
```

F#

Fonction multi-ligne

Indentation nécessaire, mais pas de `{ }`

```
let evens list =  
    let isEven x = // 📌 Sous-fonction  
        x % 2 = 0 // 💡 `=` opérateur d'égalité - Pas de `==`  
        List.filter isEven list  
  
let res = evens [1;2;3;4;5] // Vaut [2;4]
```

F#

Fonction anonyme

A.k.a. **Lambda**, arrow function

- Déclarée avec `fun` et `→`
- En général entre `()` pour question de précedence

```
let evens' list = List.filter (fun x → x % 2 = 0) list
```

F#

👉 **Note** : taille de la flèche

- Fine `→` en F#, Java
- Large / *fat* `⇒` en C#, JS

Convention de noms courts

- `x`, `y`, `z` : paramètres de type valeurs simples
- `f`, `g`, `h` : paramètres de type fonction
- `xs` : liste de `x` → `x :: xs` (ou `h :: t`) = *head* et *tail* d'une liste non vide
- `_` : *discard* / élément ignoré car non utilisé (*comme en C# 7.0*)

Bien adapté quand fonction courte ou très générique :

```
// Fonction qui renvoie son paramètre d'entrée, quel qu'il soit
let id x = x

// Composition de 2 fonctions
let compose f g = fun x → g (f x)
```

F#

Piping

Opérateur *pipe* `▷` : même idée que `|` UNIX

→ Envoyer la valeur à gauche dans une fonction à droite

→ Ordre naturel "sujet verbe" - idem appel méthode d'un objet

```
let a = 2 ▷ add 3 // Se lit comme "2 + 3"

let nums = [1;2;3;4;5]
let evens = nums ▷ List.filter (fun x → x % 2 = 0)
// Idem      List.filter (fun x → x % 2 = 0) nums
```

F#

```
// ≈ C#
var a = 2.Add(3);
var nums = new[] { 1, 2, 3, 4, 5 };
var evens = nums.Where(x ⇒ x % 2 = 0);
```

C#

Chainage de *pipes* - *Pipeline*

Comme fluent API en C# mais natif : pas besoin de méthode d'extension 👍

Façon naturelle de représenter le flux de données entre différentes opérations
→ Sans variable intermédiaire 👍

Écriture :

```
// Sur une seule ligne (courte)
let res = [1;2;3;4;5] ▷ List.filter (fun x → x % 2 = 0) ▷ List.sum

// Sur plusieurs lignes ⇒ fait ressortir les différentes opérations
let res' =
    [1; 2; 3; 4; 5]
    ▷ List.filter isOdd    // Avec `let isOdd x = x % 2 < 0`
    ▷ List.map square      // `let square x = x * x`
    ▷ List.map addOne      // `let addOne x = x + 1`
```

F#

Expression `if/then/else`

💡 `if b then x else y` \simeq Opérateur ternaire C# `b ? x : y`

```
let isEven n =  
    if n % 2 = 0 then  
        "Even"  
    else  
        "Odd"
```

F#

👉 Si `then` ne renvoie pas de valeur, `else` facultatif

```
let printIfEven n msg =  
    if n > isEven then  
        printfn msg
```

F#

Pattern matching avec `match/with`

```
let translate civility =  
    match civility with  
    | "Mister" → "Monsieur"  
    | "Madam"  → "Madame"  
    | "Miss"   → "Mademoiselle"  
    | _       → ""      // 🐞 wilcard `_`
```

F#

Équivalent en C# 8 :

```
public static string Translate(string civility) ⇒  
    civility switch {  
        "Mister" ⇒ "Monsieur"  
        "Madam"  ⇒ "Madame"  
        "Miss"   ⇒ "Mademoiselle"  
        _       ⇒ ""  
    }
```

C#

Exception

Handling Exception

→ Bloc `try/with` -- ⚠ Piège : ~~try/catch~~

```
let tryDivide x y =  
    try  
        Some (x / y)  
    with :? System.DivideByZeroException →  
        None
```

F#

Throwing Exception

→ Fonctions `failwith` et `invalidArg`

```
let fn arg = invalidArg (nameof arg) "Message ... "  
  
let divide x y =  
    if y = 0  
    then failwith "Divisor cannot be zero"  
    else x / y
```

F#

👉 Pour erreurs métier i.e. cas prévus, non exceptionnels :
Préférer type `Result` / *Railway-oriented programming* 📌

🔗 Handling Errors Elegantly <https://devonburriss.me/how-to-fsharp-pt-8/>

Ordre des déclarations

⚠ Déclarations ordonnées de haut en bas

- Déclaration précède usage
- Au sein d'un fichier
- Entre fichiers dépendants
 - 🙅 Importance de l'ordre des fichiers dans un `.fsproj`
- Bénéfice : pas de dépendances cycliques 👍

```
let result = fn 2
//           ~ ✨ Error FS0039: La valeur ou le constructeur 'fn' n'est pas défini

let fn i = i + 1 // 🙅 Doit être déclarée avant `result`
```

F#

Indentation

- Très importante pour lisibilité du code
 - Crée struct. visuelle qui reflète struct. logique / hiérarchie
 - `{ }` alignées verticalement (C#) = aide visuelle mais < indentation
- Essentielle en F# :
 - Façon de définir des blocs de code
 - Compilateur assure que indentation est correcte

👉 Conclusion :

- F# force à bien indenter
- Mais c'est pour notre bien
- Car c'est bénéfique pour lisibilité du code 👍

Ligne verticale d'indentation

- Démarre après `let ... =`, `(`, `then` / `else`, `try` / `finally`, `do`, → (dans clause `match`) mais pas `fun` !
- Commence au 1er caractère non *whitespace* qui suit
- Tout le reste du bloc doit s'aligner verticalement
- L'indentation peut varier d'un bloc à l'autre

```
let f =  
    let x=1      // ligne d'indentation fixée en column 3 (indépendamment des lignes précédentes)  
    x+1         // ➡ cette ligne (du même bloc) doit commencer en column 3 ; ni 2, ni 4 !  
  
let f = let x=1 // Indentation en column 10
```

F#

<https://fsharpforfunandprofit.com/posts/fsharp-syntax/>

Indentation - *Guideline*

- Utiliser des **espaces**, pas des ~~tabulations~~
- Utiliser **4 espaces** par indentation
 - Facilite la détection visuelle des blocs
 - ... qui ne peut se baser sur les `{ }` comme en C#

Indentation - *Guideline* (2)

- Éviter un alignement sensible au nom, a.k.a *vanity alignment*
 - Risque de rupture de l'alignement après renommage → 💥 Compilation
 - Bloc trop décalé à droite → nuit à la lisibilité

```
// 🍌 OK
```

```
let myLongValueName =  
    someExpression  
    ▷ anotherExpression
```

```
// ⚠️ À éviter
```

```
let myLongValueName = someExpression  
    ▷ anotherExpression // ➡ Dépend de la longueur de `myLongValueName`
```

F#

3 ■ Premiers concepts



Curryfication

Consiste à transformer :

- une fonction prenant N paramètres
 - `Func<T1, T2 ... Tn, TReturn>` en C#
- en une chaîne de N fonctions prenant 1 paramètre
 - `Func<T1, Func<Tn, ... Func<Tn, TReturn>>>`

Application partielle

Appel d'une fonction avec moins d'arguments que son nombre de paramètres

- Possible grâce à la curryfication
- Renvoie fonction prenant en paramètre le reste d'arguments à fournir

```
// Template à 2 paramètres
let insideTag (tagName: string) (content: string) =
    $"<{tagName}>{content}</{tagName}>"

// Helpers à 1! paramètre `content`, `tagName` étant fixé par application partielle
let emphasize = insideTag "em" // `tagName` fixé à la valeur "em"
let strong    = insideTag "strong" // `tagName` fixé à la valeur "strong"

// Equivalent - élégant mais + explicite
let em content = insideTag "em" content
```

F#

Expression vs Instruction (*Statement*)

“ Une **instruction** produit un effet de bord.
Une **expression** produit une valeur et un éventuel effet de bord (**à éviter**). ”

→ F# est un langage fonctionnel, à base **d'expressions** uniquement.

→ C# est un langage impératif, à base **d'instructions** (*statements*)
mais comporte de + en + de sucre syntaxique à base d'expressions :

→ Opérateur ternaire `b ? x : y`

→ [Null-conditional operator](#) `?.` en C# 6 : `model?.name`

→ [Null-coalescing operator](#) `??` en C# 8 : `label ?? '(Vide)'`

→ Expression lambda en C# 3 avec LINQ : `numbers.Select(x ⇒ x + 1)`

→ [Expression-bodied members](#) en C# 6 et 7

→ Expression `switch` en C# 8





Avantages des expressions / instructions

- **Concision** : code + compact == + lisible
- **Composabilité** : composer expressions == composer valeurs
 - Addition, multiplication... de nombres,
 - Concaténation dans une chaîne,
 - Collecte dans une liste...
- **Compréhension** : pas besoin de connaître les instructions précédentes
- **Testabilité** : expressions pures (*sans effet de bord*) + facile à tester
 - *Prédictible* : même inputs produisent même outputs
 - *Isolée* : phase *arrange/setup* allégée (*pas de mock...*)

En F# « Tout est expression »

- Une fonction se déclare et se comporte comme une valeur
 - En param ou en sortie d'une autre fonction (*dite high-order function*)
- Éléments du *control flow* sont aussi des expressions
 - Branches des `if/else` et `match/with` (`~=switch`) renvoient une valeur.
 - Sauf bloc `for` mais style impératif, pas fonctionnel → cas particuliers

Remarques

-  `let a = expression` ressemble à instruction de déclaration/affectation
-  **Débogage** : quasi même expérience de débogage pas à pas qu'en C#

Tout est expression • Conséquences

Pas de `void`

- Remplacé par le type `unit` ayant 1! valeur notée `()` 
- `else` optionnel si `if` renvoie `unit`

Pas de *early exit*

- Pas de `return` pour court-circuiter fonction 
- Pas de `break` pour sortir d'une boucle `for/while` 

Pas de *early exit* - Solutions

- 🤮 `throw BreakException` (cf. [réponse StackOverflow](#))
- 😞 Impératif: `while (ok)` avec `ok` mutable
- ✅ Fonctionnel via fonction récursive ⚠️
 - *Décide ou non de continuer la "boucle" en s'appelant*

```
let rec firstOr defaultValue predicate list =  
    match list with  
    | [] → defaultValue // ➡ Sortie  
    | x :: _ when predicate x → x // ➡ Sortie  
    | _ :: rest → firstOr defaultValue predicate rest // ➡ Appel récursif → continue  
  
let test1 = firstOr -1 (fun x → x > 5) [1] // -1  
let test2 = firstOr -1 (fun x → x > 5) [1; 6] // 6
```

F#

Pas de *early exit* - Inconvénients

⚠ Risque de `if` imbriqués (+ difficile à comprendre)

💡 Solutions (les mêmes qu'en C#):

- Décomposer en sous fonctions -- "*Do one thing*" de *Clean Code*
- Aplatir : réunir valeurs + *Pattern matching*
 - `match x, y with ...` où `x, y` est un tuple 📌

Typage, inférence et cérémonie

Poids de la cérémonie \neq Force du typage

→ Cf. <https://blog.ploeh.dk/2019/12/16/zone-of-ceremony/>

Lang	Force du typage	Inférence	Cérémonie
JS	Faible (dynamique)	×	Faible
C#	Moyen (statique nominal)	Faible	Fort
TS	Fort (statique structurel + ADT)	Moyenne	Moyen
F#	Fort (statique nominal + ADT)	Élevée	Faible

ADT = *Algebraic Data Types* = product types + sum types

Inférence de type

Objectif : Typer explicitement le moins possible

- Moins de code à écrire 👍
- Compilateur garantit la cohérence
- IntelliSense aide le codage et la lecture
- Importance du nommage pour lecture hors IDE ⚠

Inférence de type en C# : plutôt faible

- Déclaration d'une méthode → paramètres et retour ❌
- Argument lambda : `list.Find(i ⇒ i = 5)` ✓
- Variable, y.c. objet anonyme : `var o = new { Name = "John" }` ✓
 - Sauf lambda : `Func<int, int> fn = (x: int) ⇒ x + 1;` → KO avec `var`
 - 💡 LanguageExt : `var fn = fun((x: int) ⇒ x + 1);` ✓
- Initialisation d'un tableau : `new[] { 1, 2 }` ✓
- Appel à une méthode générique avec argument, sauf constructeur :
 - `Tuple.Create(1, "a")` ✓
 - `new Tuple<int, string>(1, "a")` ❌
- C# 9 *target-typed expression* `StringBuilder sb = new();` ✓

Inférence en TypeScript - *The good parts* 👍

👉 Code pur JavaScript (*modulo* `as const` qui reste élégant)

```
const obj1 = { a: 1 }; // { a: number }
const obj2 = Object.freeze({ a: 1 }); // { readonly a: number }
const obj3 = { a: 1 } as const; // { readonly a: 1 }

const arr1 = [1, 2, null]; // (number | null)[]
const arr2 = [1, 2, 3]; // number[]
const arr3 = arr2.map(x => x * x); // ✓ Pure lambda

// Type littéral
let s = 'a'; // string
const a = 'a'; // "a"
```

TS

Inférence en TypeScript - Limites

```
// 1. Combinaison de littéraux
const a = 'a'; // "a"
const aa = a + a; // string (et pas "aa")

// 2. Tuple, immuable ou non
const tupleMutableKo = [1, 'a']; // ✗ (string | number)[]
const tupleMutableOk: [number, string] = [1, 'a'];

const tupleImmutKo = Object.freeze([1, 'a']); // ✗ readonly (string | number)[]
const tupleImmutOk = [1, 'a'] as const; // readonly [1, "a"]

// 3. Paramètres d'une fonction → gêne *Extract function* 😞
// ⇒ Refacto de `arr2.map(x ⇒ x * x)` en `arr2.map(square)`
const square = x ⇒ x * x; // ✗ Sans annotation
// ~ Parameter 'x' implicitly has an 'any' type.(7006)
const square = (x: number) ⇒ x * x; // (x: number) ⇒ number
```

TS

Inférence de type en F# : forte 💪

Méthode [Hindley-Milner](#)

- Capable de déduire le type de variables, expressions et fonctions d'un programme dépourvu de toute annotation de type
- Se base sur implémentation et usage

```
let helper instruction source =  
    if instruction = "inc" then // 1. `instruction` a même type que `"inc"` ⇒ `string`  
        source + 1             // 2. `source` a même type que `1` ⇒ `int`  
    elif instruction = "dec" then  
        source - 1  
    else  
        source                 // 3. `return` a même type que `source` ⇒ `int`
```

F#

Inférence en F# - Généralisation automatique

```
// Valeurs génériques
let a = [] // 'a list

// Fonctions génériques : 2 param 'a, renvoie 'a list
let listOf2 x y = [x; y]

// Idem avec 'a "comparable"
let max x y = if x > y then x else y
```

F#

- 🙌 En F#, type générique précédé d'une apostrophe : `'a`
- Partie `when 'a : comparison` = contraintes sur type
- 💡 Généralisation rend fonction utilisable dans + de cas 🎉
- `max` utilisable pour 2 args de type `int`, `float`, `string`...
- 🙌 D'où l'intérêt de laisser l'inférence plutôt que d'annoter les types

Inférence en F# - Résolution statique

Problème : type inféré + restreint qu'attendu 🤔

```
let sumOfInt x y = x + y // Seulement int
```

F#

→ Juste `int` ? Pourtant `+` marche pour les nombres et les chaînes 😞

Solution : fonction `inline`

```
let inline sum x y = x + y // Full generic: 2 params ^a ^b, retour ^c
```

F#

→ Paramètres ont un type résolu statiquement = à la compilation

→ Noté avec un *caret* : `^a`

→ ≠ Type générique `'a`, résolu au runtime

Inférence en F# - Limites

⚠ Type d'un objet non inférable depuis ses méthodes

```
F#
let helperKo instruction source = // ✨ Error FS0072: Recherche d'un objet de type indéterminé ...
    match instruction with
    | 'U' → source.ToUpper()
    | _   → source

let helper instruction (source: string) = [ ... ] // ➡ Annotation nécessaire

let info list = if list.Length = 0 then "Vide" else "..." // ✨ Error FS0072 ...
```

👉 D'où l'intérêt de l'approche FP (*fonctions séparées des données*)
Vs approche OO (*données + méthodes ensemble dans objet*)

Inférence en F# - Gestion de la précedence

⚠ Ordre des termes impacte inférence

```
let listKo = List.sortBy (fun x → x.Length) ["three"; "two"; "one"]  
// ✨ Error FS0072: Recherche d'un objet de type indéterminé ...
```

F#

💡 Solutions

1. Inverser ordre des termes en utilisant le *pipe*

```
let listOk = ["three"; "two"; "one"] ▷ List.sortBy (fun x → x.Length)
```

F#

2. Utiliser fonction plutôt que méthode

```
let listOk' = List.sortBy String.length ["three"; "two"; "one"]
```

F#

4 ■ Quiz




1. Qui est le papa de F# ?

A. Anders Hejlsberg

B. Don Syme

C. Scott Wlaschin

 10''



1. Qui est le papa de F# ?

A. Anders Hejlsberg ✗

Papa de C# et de TypeScript

B. Don Syme ✓

 social-network w:30 [dsymetweets](#) •  [F# Code I Love](#)

C. Scott Wlaschin ✗

Auteur du blog [F# for Fun and Profit](#), mine d'or pour F#



2. Comment se nomme l'opérateur `::` ?

A. Append

B. Concat

C. Cons

 10''



2. Comment se nomme l'opérateur `::` ?

A. Append ❌

`List.append` : concatène 2 listes

B. Concat ❌

`List.concat` : concatène un ensemble de listes

C. Cons ✅

`newItem :: list` est la manière la + rapide de créer une nouvelle liste avec un nouvel élément en tête : `1 :: [2; 3]` renvoie `[1; 2; 3]`.



3. Cherchez l'intrus

A. `let a = "a"`

B. `let a () = "a"`

C. `let a = fun () → "a"`

🕒 15''



3. Cherchez l'intrus

B et C sont des fonctions, A est juste une `string`.

A. `let a = "a"` ✓

B. `let a () = "a"` ✗

C. `let a = fun () → "a"` ✗



4. Quelle ligne ne compile pas ?

```
let evens list =  
    let isEven x =  
        x % 2 = 0  
    List.filter isEven list
```

F#

Ligne 1. `let evens list =`

Ligne 2. `let isEven x =`

Ligne 3. `x % 2 = 0`

Ligne 4. `List.filter isEven list`

🕒 15''



4. Quelle ligne ne compile pas ?

Ligne 3. `x % 2 = 0`

Problème d'indentation

```
let evens list =  
    let isEven x =  
        x % 2 = 0 // ➡ Manque une indentation ici  
        List.filter isEven list
```

F#



5. Comment se nomme l'opérateur ?

A. Compose

B. Chain

C. Pipeline

D. Pipe

 10''



5. Comment se nomme l'opérateur ?

A. Compose 

L'opérateur de composition est  

B. Chain 

C. Pipeline 

D. Pipe 



6. Quelle expression compile ?

A. `a = "a" && b ≠ "*"`

B. `a = "a" && b ◇ "*"`

C. `a = "a" && b ◇ "*"`

D. `a = "a" && b ≠ "*"`

🕒 15"



6. Quelle expression compile ?

👉 En F#, les opérateurs d'égalité et d'inégalité sont respectivement `=` et `<>`.

A. `a = b && b ≠ ""` ❌

B. `a = b && b <> ""` ❌

C. `a = b && b <> ""` ✅

D. `a = b && b ≠ ""` ❌



5. ■ Le Récap'



Récap'

- Syntaxe du F#
 - Aperçu général, déjà copieux
 - Nous permettra de nous focaliser sur des détails
- Concepts engrammés dans F#
 - Curryfication, application partielle
 - « Une expression sinon rien ! »
 - Inférence de type



Complément

<https://blog.ploeh.dk/2015/08/17/when-x-y-and-z-are-great-variable-names>

En F#, les fonctions et variables ont souvent des noms courts : `f`, `x` et `y`.
Mauvais nommage ? Non, pas dans les cas suivants :

- Fonction hyper générique → paramètres avec nom générique
- Portée courte → code + lisible avec nom court que nom long

Merci 🙏

SOAT

→ Digitalize society



SOAT.FR