



→ Digitalize society



# Formation F# 5.0

## *Pattern matching*



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



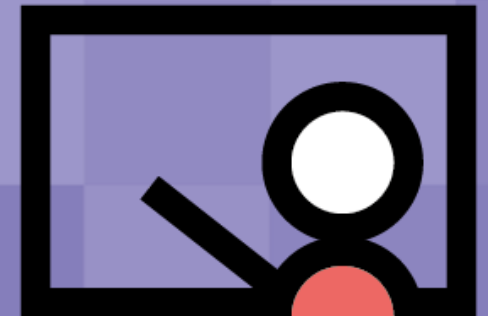
DeneauRomain



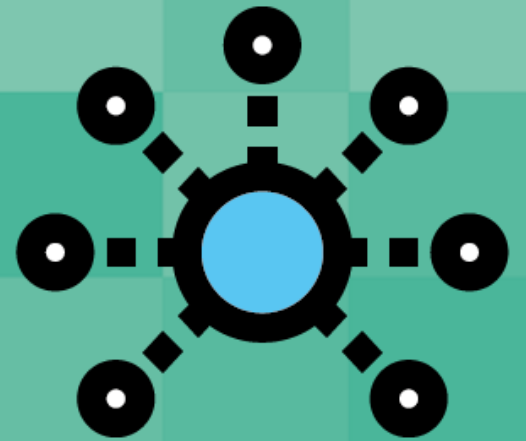
rdeneau

# Sommaire

- Patterns
- Match expression
- Active patterns



# 1. Généralités : *Patterns*



# Patterns

*Patterns* = règles pour détecter la structure de données en entrée

Utilisés abondamment en F#

- Dans *match expression*, *let binding* de valeurs et de paramètres de fonctions
- Très pratiques pour manipuler les types algébriques F# (tuple, record, union)
- Composables : supporte plusieurs niveaux d'imbrication
- Assemblables par ET/OU logiques
- Supporte les littéraux : `1.0`, `"test"` ...

# Wildcard Pattern

Représenté par `_`, seul ou combiné avec un autre *pattern*

Toujours vrai

→ A placer en dernier dans une *match expression*

⚠ Toujours chercher en 1er à traiter exhaustivement/explicitement tous les cas  
Quand impossible, utiliser alors le `_`

```
match option with
| Some 1 → ...
| _ → ...           // ⚠ Non exhaustif

match option with
| Some 1 → ...
| Some _ | None → ... // 👉 \+ exhaustif
```

F#

# Constant Pattern

Détecte constantes, `null` et littéraux de nombre, `char`, `string`, `enum`

```
[<Literal>]
let Three = 3 // Constante

let is123 num = // int → bool
    match num with
    | 1 | 2 | Three → true
    | _ → false
```

F#

## 👉 Notes :

- Le pattern de `Three` est aussi classé en tant que *Identifier Pattern* 📌
- Pour le matching de `null`, on parle aussi de *Null Pattern*

# Identifier Pattern

Détecte les *cases* d'un type union ainsi que leur éventuel contenu

```
type PersonName =  
    | FirstOnly of string  
    | LastOnly  of string  
    | FirstLast of string * string  
  
let classify personName =  
    match personName with  
    | FirstOnly _   → "First name only"  
    | LastOnly  _   → "Last name only"  
    | FirstLast _   → "First and last names"
```

F#



# Variable Pattern

Assigne la valeur détectée à une "variable" pour l'utiliser ensuite

Exemple : variables `firstName` et `lastName` ci-dessous

```
type PersonName =  
    | FirstOnly of string  
    | LastOnly  of string  
    | FirstLast of string * string  
  
let confirm personName =  
    match personName with  
    | FirstOnly (firstName) → printf "May I call you %s?" firstName  
    | LastOnly  (lastName) → printf "Are you Mr. or Ms. %s?" lastName  
    | FirstLast (firstName, lastName) → printf "Are you %s %s?" firstName lastName
```

F#

# Variable Pattern (2)

⚠ On ne peut pas lier à plusieurs reprises vers la même variable

```
let elementsAreEqualKo tuple =  
    match tuple with  
    | (x,x) → true // ✨ Error FS0038: 'x' est lié à deux reprises dans ce modèle  
    | (_,_) → false
```

F#

Solutions : utiliser 2 variables puis vérifier l'égalité

```
// 1. Guard clause  
let elementsAreEqualOk = function  
    | (x,y) when x = y → true  
    | (_,_) → false  
  
// 2. Déconstruction  
let elementsAreEqualOk' (x, y) = x = y
```

F#

# Champs nommés de *case* d'union

Plusieurs possibilités :

- ① Pattern "anonyme" du tuple complet
- ② Pattern d'un seul champ par son nom → `Field = value`
- ③ Pattern de plusieurs champs par leur nom → `F1 = v1; F2 = v2`

```
type Shape =  
    | Rectangle of Height: int * Width: int  
    | Circle of Radius: int  
  
let describe shape =  
    match shape with  
    | Rectangle (0, _)                                // ①  
    | Rectangle (Height = 0)                          → "Flat rectangle"           // ②  
    | Rectangle (Width = w; Height = h) → $"Rectangle {w} × {h}"           // ③  
    | Circle radius                                → $"Circle Ø {2*radius}"
```

F#

# Alias Pattern

`as` permet de nommer un élément dont le contenu est déconstruit

```
let (x, y) as coordinate = (1, 2)
printfn "%i %i %A" x y coordinate // 1 2 (1, 2)
```

F#

💡 Marche aussi dans les fonctions :

```
type Person = { Name: string; Age: int }

let acceptMajorOnly ({ Age = age } as person) =
    if age < 18 then None else Some person
```

F#

# OR et AND Patterns

Permettent de combiner deux patterns (*nommés `P1` et `P2` ci-après*)

- `P1 | P2` → P1 ou P2. Ex : `Rectangle (0, _) | Rectangle (_, 0)`
- `P1 & P2` → P1 et P2. Utilisé surtout avec *active patterns* !

💡 Utiliser la même variable :

```
type Upload = { Filename: string; Title: string option }  
  
let titleOrFile ({ Title = Some name } | { Filename = name }) = name  
  
titleOrFile { Filename = "Report.docx"; Title = None }           // Report.docx  
titleOrFile { Filename = "Report.docx"; Title = Some "Report+" } // "Report+"
```

F#

# Parenthesized Pattern

Usage des parenthèses `()` pour grouper des patterns, pour gérer la précedence

```
type Shape = Circle of Radius: int | Square of Side: int
let countFlatShapes shapes =
    let rec loop rest count =
        match rest with
        | (Square (Side = 0) | (Circle (Radius = 0))) :: tail → loop tail (count + 1) // ①
        | _ :: tail → loop tail count
        | [] → count
    loop shapes 0
```

F#

👉 **Note** : la ligne ① ne compilerait sans faire `() :: tail`

# Parenthesized Pattern (2)

- ⚠ Les parenthèses compliquent la lecture
- 💡 Essayer de s'en passer quand c'est possible

```
let countFlatShapes shapes =  
    let rec loop rest count =  
        match rest with  
        | Circle (Radius = 0) :: tail  
        | Square (Side = 0) :: tail  
        → loop tail (count + 1)  
    // [ ... ]
```

F#

# Construction Patterns

Reprennent syntaxe de construction d'un type pour le déconstruire

- Cons et List Patterns
- Array Pattern
- Tuple Pattern
- Record Pattern



# Cons et List Patterns

≈ Inverses de 2 types de construction d'une liste, avec même syntaxe

*Cons Pattern* : `head :: tail` → décompose une liste (avec  $\geq 1$  élément) en :

- *Head* : 1er élément
- *Tail* : autre liste avec le reste des éléments - peut être vide

*List Pattern* : `[items]` → décompose une liste en 0..N éléments

- `[]` : liste vide
- `[x]` : liste avec 1 élément mis dans la variable `x`
- `[x; y]` : liste avec 2 éléments mis dans les variables `x` et `y`
- `[_; _]` : liste avec 2 éléments ignorés

💡 `x :: []`  $\equiv$  `[x]`, `x :: y :: []`  $\equiv$  `[x; y]` ...

## Cons et List Patterns (2)

La *match expression* par défaut combine les 2 patterns :

→ Une liste est soit vide `[]`, soit composée d'un item et du reste : `head :: tail`

Les fonctions récursives parcourant une liste utilise le pattern `[]` pour stopper la récursion :

```
let rec printList l =  
    match l with  
    | head :: tail →  
        printf "%d " head  
        printList tail           // Récursion sur le reste  
    | [] → printfn ""           // Fin de récursion : liste parcourue entièrement
```

F#

# Array Pattern

Syntaxe: `[] items []` pour 0..N items entre `;`

```
let length vector =  
    match vector with  
    | [] x [] → x  
    | [] x; y [] → sqrt (x*x + y*y)  
    | [] x; y; z [] → sqrt (x*x + y*y + z*z)  
    | _ → invalidArg (nameof vector) $"Vector with more than 4 dimensions not supported"
```

F#

👉 Il n'existe pas de pattern pour les séquences, vu qu'elles sont *"lazy"*.

# Tuple Pattern

Syntaxe : `items` ou `(items)` pour 2..N items entre `,`

💡 Pratique pour pattern matcher plusieurs valeurs en même temps

```
type Color = Red | Blue
type Style = Background | Text

let css color style =
    match color, style with
    | Red, Background → "background-color: red"
    | Red, Text → "color: red"
    | Blue, Background → "background-color: blue"
    | Blue, Text → "color: blue"
```

F#

# Record Pattern

Syntaxe : `{ Fields }` pour 1..N `Field = variable` entre `;`

→ Pas obligé de spécifier tous les champs du Record

→ En cas d'ambiguïté, qualifier le champ : `Record.Field`

💡 Marche aussi pour les paramètres d'une fonction :

```
type Person = { Name: string; Age: int }
```

F#

```
let displayMajority { Age = age; Name = name } =  
    if age ≥ 18  
    then printfn "%s is major" name  
    else printfn "%s is minor" name
```

```
let john = { Name = "John"; Age = 25 }  
displayMajority john // John is major
```

# Record Pattern (2)

⚠ **Rappel** : il n'y a pas de pattern pour les *Records* anonymes !

```
type Person = { Name: string; Age: int }  
  
let john = { Name = "John"; Age = 25 }  
let { Name = name } = john // 🔥 val name : string = "John"  
  
let john' = [{| john with Civility = "Mister" |}]  
let [{| Name = name' |}] = john' // ✨
```

F#

# Type Test Pattern

Syntaxe : `? : Subtype`

Usage : avec une hiérarchie de types

```
open System.Windows.Forms

let RegisterControl (control: Control) =
    match control with
    | :? Button as button → button.Text ← "Registered."
    | :? CheckBox as checkbox → checkbox.Text ← "Registered."
    | :? Windows → invalidArg (nameof control) "Window cannot be registered"
    | _ → ()
```

F#

# Type Test Pattern - Bloc `try` / `with`

On rencontre fréquemment ce pattern dans les blocs `try` / `with` :

```
try
    printfn "Difference: %i" (42 / 0)
with
| :? DivideByZeroException as x →
    printfn "Fail! %s" x.Message
| :? TimeoutException →
    printfn "Fail! Took too long"
```

F#



# Type Test Pattern - Boxing

Le *Type Test Pattern* ne marche qu'avec des types références.

→ Pour un type valeur ou inconnu, il faut le convertir en objet (*a.k.a boxing*)

```
let isIntKo = function :? int → true | _ → false
// ✨ Error FS0008: test de type au moment de l'exécution du type 'a en int...
```

F#

```
let isInt x =
    match box x with
    | :? int → true
    | _ → false
```

# 2. *Match Expression*



# Match expression

Similaire à une expression `switch` en C# 8.0 en + puissant grâce aux patterns

Syntaxe :

```
match test-expression with
| pattern1 [ when condition ] → result-expression1
| pattern2 [ when condition ] → result-expression2
| ...
```

F#

Renvoie le résultat de la 1ère branche dont le pattern "match" `test-expression`

👉 **Note** : toutes les branches doivent renvoyer le même type !

# Match expression - Exhaustivité

Un `switch` doit toujours définir un cas par défaut (*cf. pattern wildcard `_`*)

Sinon : warning à la compilation, risque de  `MatchFailureException` au runtime

Pas nécessaire dans une *match expression* si les branches couvrent tous les cas car le compilateur vérifie leur exhaustivité et les branches "mortes"

```
let fn x =  
  match x with  
  | Some true  → "ok"  
  | Some false → "ko"  
  | None       → ""  
  | _          → "?" // ⚠ Warning FS0026: Cette règle n'aura aucune correspondance
```

F#

# Match expression - Exhaustivité (2)

👉 **Conseil** : + les branches sont exhaustives, + le code est explicite et sûr

Exemple : matcher tous les cases d'un type union permet de gérer l'ajout d'un case par un warning à la compilation :

```
Warning FS0025: Critères spéciaux incomplets dans cette expression
```

- Détection d'un ajout accidentel
- Identification du code à changer pour gérer le nouveau case

# Match expression - Guard

Syntaxe : `pattern1 when condition`

Usage : raffiner un pattern, via contraintes sur des variables

```
let classifyBetween low top value =  
    match value with  
    | x when x < low → "Inf"    // 💡 Alternative : `_ when value < low`  
    | x when x = low → "Low"  
    | x when x = top → "Top"  
    | x when x > top → "Sup"  
    | _ → "Between"
```

```
let test1 = 1 ▷ classifyBetween 1 5 // "Low"  
let test2 = 6 ▷ classifyBetween 1 5 // "Sup"
```

💡 La *guard* n'est évaluée que si le pattern est satisfait.

# Match expression - Guard et Pattern OR

Le pattern OR a une *precedence/priorité* plus élevée que la *guard*:

```
type Parity = Even of int | Odd of int

let parityOf value =
    if value % 2 = 0 then Even value else Odd value

let hasSquare square value =
    match parityOf square, parityOf value with
    | Even x2, Even x
    | Odd x2, Odd x
        when x2 = x*x → true // ➡ Porte sur les 2 patterns précédents
    | _ → false

let test1 = 2 ▷ hasSquare 4 // true
let test2 = 3 ▷ hasSquare 9 // true
```

F#

# Match function

Syntaxe :

```
function
| pattern1 [ when condition ] → result-expression1
| pattern2 [ when condition ] → result-expression2
| ...
```

F#

Equivalent à une lambda prenant un paramètre implicite qui est "matché" :

```
fun value →
    match value with
    | pattern1 [ when condition ] → result-expression1
    | pattern2 [ when condition ] → result-expression2
    | ...
```

F#



# Match function - Intérêts

## 1. Dans pipeline

```
value
▷ is123
▷ function
  | true  → "ok"
  | false → "ko"
```

F#

## 2. Écriture + succincte d'une fonction

```
// ⚠ Paramètre implicite ⇒ peut rendre le code \+ difficile à comprendre !
let is123 = function
  | 1 | 2 | 3 → true
  | _ → false
```

F#

# Match function - Limites

⚠ Paramètre implicite => peut rendre le code + difficile à comprendre !

Exemple : fonction déclarée avec d'autres paramètres eux explicites  
→ On peut se tromper sur le nombre de paramètres et leur ordre :

```
let classifyBetween low high = function // ➡ 3 paramètres : low, high + 1 implicite
| x when x < low   → "Inf"
| x when x = low   → "Low"
| x when x = high  → "High"
| x when x > high  → "Sup"
| _               → "Between"

let test1 = 1 ▷ classifyBetween 1 5 // "Low"
let test2 = 6 ▷ classifyBetween 1 5 // "Sup"
```

F#

# Fonction `fold`

Fonction associée à un type union et masquant la logique de *matching*  
Prend N+1 paramètres pour un type union avec N cases `CaseI of 'DataI`  
→ N fonctions `'DataI → 'T` (1 / case), avec `'T` le type de retour de `fold`  
→ En dernier, l'instance du type union à matcher

```
type [<Measure>] C
type [<Measure>] F

type Temperature =
| Celsius      of float<C>
| Fahrenheit  of float<F>

module Temperature =
    let fold mapCelsius mapFahrenheit temperature : 'T =
        match temperature with
        | Celsius x      → mapCelsius x      // mapCelsius : float<C> → 'T
        | Fahrenheit x → mapFahrenheit x    // mapFahrenheit: float<F> → 'T
```

F#

# Fonction **fold** : utilisation

```
module Temperature =  
    // ...  
    let [<Literal>] FactorC2F = 1.8<F/C>  
    let [<Literal>] DeltaC2F = 32.0<F>  
  
    let celsiusToFahrenheit x = (x * FactorC2F) + DeltaC2F // float<C> → float<F>  
    let fahrenheitToCelsius x = (x - DeltaC2F) / FactorC2F // float<F> → float<C>  
  
    let toggleUnit temperature =  
        temperature ▷ fold  
            (celsiusToFahrenheit >> Fahrenheit)  
            (fahrenheitToCelsius >> Celsius)  
  
let t1 = Celsius 100.0<C>  
let t2 = t1 ▷ Temperature.toggleUnit // Fahrenheit 212.0
```

F#

# Fonction `fold` : intérêt

`fold` masque les détails d'implémentation du type

Par exemple, on peut ajouter un *case* `Kelvin` et n'impacter que `fold`, pas les fonctions qui l'appellent comme `toggleUnit` :

```
type [<Measure>] C
type [<Measure>] F
type [<Measure>] K // 🌟

type Temperature =
| Celsius      of float<C>
| Fahrenheit  of float<F>
| Kelvin      of float<K> // 🌟

// ...
```

F#

# Fonction **fold** : intérêt (2)

```
// ...  
module Temperature =  
    let fold mapCelsius mapFahrenheit temperature : 'T =  
        match temperature with  
        | Celsius x      → mapCelsius x          // mapCelsius: float<C> → 'T  
        | Fahrenheit x  → mapFahrenheit x       // mapFahrenheit: float<F> → 'T  
        | Kelvin x      → mapCelsius (x * 1.0<C/K> + 273.15<C>) // 🌞  
  
Kelvin 273.15<K>  
▷ Temperature.toggleUnit  
▷ Temperature.toggleUnit  
// Celsius 0.0<C>
```

F#

# 3 ■ *Active Patterns*



# Limitations du *Pattern Matching*

Nombre limité de patterns

Impossibilité de factoriser l'action de patterns avec leur propre guard

→ `Pattern1 when Guard1 | Pattern2 when Guard2 → do` 

→ `Pattern1 when Guard1 → do | Pattern2 when Guard2 → do` 

Patterns ne sont pas des citoyens de 1ère classe

*Ex : une fonction ne peut pas renvoyer un pattern*

→ Juste une sorte de sucre syntaxique

Patterns interagissent mal avec un style OOP



# Origine des *Active Patterns*

“  [Extensible pattern matching via a lightweight language extension](#) ”

 Publication de 2007 de Don Syme, Gregory Neverov, James Margetson ”

Intégré à F# 2.0 (2010)

## Idées

- Permettre le *pattern matching* sur d'autres structures de données
- Faire de ces nouveaux patterns des citoyens de 1ère classe

# Active Patterns - Syntaxe

Syntaxe générale : `let (|Cases|) [arguments] valueToMatch = expression`

1. **Fonction** avec un nom spécial défini dans une "banane" `(| ... |)`
2. Ensemble de 1..N **cases** où ranger `valueToMatch`

💡 Sorte de fonction *factory* d'un **type union** "anonyme", défini *inline*

# Active Patterns - Types

Il existe 4 types d'active patterns :

1. Pattern total simple
2. Pattern total multiple
3. Pattern partiel
4. Pattern paramétré

💡 *Partiel* et *total* indique la faisabilité du « rangement dans le(s) case(s) » de la valeur en entrée

- **Partiel** : il n'existe pas toujours une case correspondante
- **Total** : il existe forcément une case correspondante → pattern exhaustif

# Active pattern total simple

*A.k.a Single-case Total Pattern*

Syntaxe : `let (|Case|) [ ... parameters] value = Case [data]`

Usage : déconstruction en ligne

```
// Avec paramètre ⇒ pas très lisible 😞
let (|Default|) = Option.defaultValue // 'T → 'T option → 'T

let (Default "unknown" name1) = Some "John" // name1 = "John"
let (Default "unknown" name2) = None        // name2 = "unknown"

// Sans paramètre ⇒ mieux 👉
let (|ValueOrUnknown|) = Option.defaultValue "unknown" // 'T option → 'T

let (ValueOrUnknown name1) = Some "John" // name1 = "John"
let (ValueOrUnknown name2) = None        // name2 = "unknown"
```

F#

# Active pattern total simple (2)

Autre exemple : extraction de la forme polaire d'un nombre complexe

```
open System.Numerics

let (|Polar|) (x : Complex) =
    Polar (x.Magnitude, x.Phase)

let multiply (Polar (m1, p1)) (Polar (m2, p2)) = // Complex → Complex → Complex
    Complex(m1 + m2, p1 + p2)
```

F#

Sans l'active pattern, c'est un autre style mais de lisibilité équivalente :

```
let multiply x y =
    Complex (x.Magnitude + y.Magnitude, x.Phase + y.Phase)
```

F#

# Active pattern total multiple

*A.k.a Multiple-case Total Pattern*

Syntaxe : `let (|Case1| ... |CaseN|) value = CaseI [dataI]`

👉 Pas de paramètre possible !

```
// Ré-écriture d'un exemple précédent

// ✗ type Parity = Even of int | Odd of int
// ✗ let parityOf value = if value % 2 = 0 then Even value else Odd value

let (|Even|Odd|) x = // int → Choice<int, int>
    if x % 2 = 0 then Even x else Odd x

let hasSquare square value =
    // ✗ match parityOf square, parityOf value with
    match square, value with
    | Even x2, Even x | Odd x2, Odd x when x2 = x*x → true
    | _ → false
```

F#

# Active pattern partiel

Syntaxe : `let (|Case|_|) value = Some Case | Some data | None`

→ Renvoie type `'T option` si *Case* comprend des données, sinon `unit option`

→ Pattern matching est non exhaustif → il faut un cas par défaut

```
let (|Integer|_|) (x: string) = // (x: string) → int option
    match System.Int32.TryParse x with
    | true, i → Some i
    | false, _ → None

let (|Float|_|) (x: string) = // (x: string) → float option
    match System.Double.TryParse x with
    | true, f → Some f
    | false, _ → None

let detectNumber = function
    | Integer i → $"Integer {i}"           // detectNumber "10"
    | Float f → $"Float {f}"              // detectNumber "1,1" = "Float 1,1" (en France)
    | s → $"NaN {s}"                     // detectNumber "abc" = "NaN abc"
```

F#

# Active pattern partiel paramétré

Syntaxe : `let (|Case|_|) ... arguments value = Some Case | Some data | None`

Exemple 1 : année bissextile = multiple de 4 mais pas 100 sauf 400

```
let (|DivisibleBy|_|) factor x = // (factor: int) → (x: int) → unit option
    match x % factor with
    | 0 → Some DivisibleBy
    | _ → None

let isLeapYear year = // (year: int) → bool
    match year with
    | DivisibleBy 400 → true
    | DivisibleBy 100 → false
    | DivisibleBy 4 → true
    | _ → false
```

F#



# Active pattern partiel paramétré (2)

Exemple 2 : Expression régulière

```
let (|Regexp|_|) pattern value = // string → string → string list option
    let m = System.Text.RegularExpressions.Regex.Match(value, pattern)
    if not m.Success || m.Groups.Count < 1 then
        None
    else
        [ for g in m.Groups → g.Value ]
        ▷ List.tail // drop "root" match
        ▷ Some
```

F#

# Active pattern partiel paramétré (3)

Exemple : Couleur hexadécimale

```
let hexToInt hex = // string → int // E.g. "FF" → 255
    System.Int32.Parse(hex, System.Globalization.NumberStyles.HexNumber)

let (|HexaColor|_|) = function // string → (int * int * int) option
    // 💡 Utilise l'active pattern précédent
    // 💡 La Regex recherche 3 groupes de 2 chars étant un chiffre ou une lettre A..F
    | Regexp "#([0-9A-F]{2})([0-9A-F]{2})([0-9A-F]{2})" [ r; g; b ] →
        Some < HexaColor ((hexToInt r), (hexToInt g), (hexToInt b))
    | _ → None

match "#0099FF" with
| HexaColor (r, g, b) → $"RGB: {r}, {g}, {b}"
| otherwise → $"'{otherwise}' is not a hex-color"
// "RGB: 0, 153, 255"
```

F#

# Récap' des types d'active patterns

Type	Syntaxe	Signature
Total multiple	<code>let (   Case1   ...   CaseN   ) x</code>	<code>'T → Choice&lt;'U1, ..., 'Un&gt;</code>
Total simple	<code>let (   Case   ) x</code>	<code>'T → 'U</code>
Partiel simple	<code>let (   Case   _   ) x</code>	<code>'T → 'U option</code>
... paramétré	<code>let (   Case   _   ) p1 ... pN x</code>	<code>'P1 → ... → 'Pn → 'T → 'U option</code>

# Comprendre un active pattern

“ Comprendre comment utiliser un active pattern...  
peut s'avérer un vrai **jonglage intellectuel** ! ”

👉 Explications en utilisant les exemples précédents

# Comprendre un active pattern total

- Active pattern total  $\simeq$  Fonction *factory* d'un type union "anonyme"
- Usage : idem pattern matching d'un type union normal

```
// Single-case
let (|Cartesian|) (x: Complex) = Cartesian (x.Real, x.Imaginary)

let Cartesian (r, i) = Complex (1.0, 2.0) // r = 1.0, i = 2.0

// Double-case
let (|Even|Odd|) x = if x % 2 = 0 then Even else Odd

let parityOf = function // int → string
    | Even → "Pair"
    | Odd  → "Impair"
```

F#

# Comprendre un active pattern partiel

👉 Bien distinguer les éventuels paramètres des éventuelles données

Examiner la signature de l'active pattern : `[ ... params → ] value → 'U option`

- Les 1..N-1 paramètres = paramètres de l'active pattern
- Son retour : `'U option` → données de type `'U` ; si `'U` = `unit` → pas de donnée

À l'usage : `match value with Case [params] [data]`

- `Case params`  $\simeq$  **application partielle**, donnant active pattern sans paramètre
- `CaseWithParams data`  $\simeq$  déconstruction d'un case de type union

# Comprendre un active pattern partiel (2)

1. `let (|Integer|_|) (s: string) : int option`  
→ Usage `match s with Integer i`, avec `i: int` donnée en sortie
2. `let (|DivisibleBy|_|) (factor: int) (x: int) : unit option`  
→ Usage `match year with DivisibleBy 400`, avec `400` le paramètre `factor`
3. `let (|Regex|_|) (pattern: string) (value: string) : string list option`  
→ Usage `match s with Regex "#([0-9 ... )" [ r; g; b ]`  
→ Avec `"#([0-9 ... )"` le paramètre `pattern`  
→ Et `[ r; g; b ]` la liste en sortie décomposée en 3 chaînes

# Exercice : fizz buzz avec active pattern

Ré-écrire ce fizz buzz en utilisant un active pattern `DivisibleBy`

```
let isDivisibleBy factor number =  
    number % factor = 0  
  
let fizzBuzz = function  
    | i when i ▷ isDivisibleBy 15 → "FizzBuzz"  
    | i when i ▷ isDivisibleBy 3  → "Fizz"  
    | i when i ▷ isDivisibleBy 5  → "Buzz"  
    | other → string other  
  
[1..15] ▷ List.map fizzBuzz  
// ["1"; "2"; "Fizz"; "4"; "Buzz"; "Fizz";  
//  "7"; "8"; "Fizz"; "Buzz"; "11";  
//  "Fizz"; "13"; "14"; "FizzBuzz"]
```

F#



# Fizz buzz avec active pattern : solution

```
let isDivisibleBy factor number =  
    number % factor = 0  
  
let (|DivisibleBy|_|) factor number =  
    if number ▷ isDivisibleBy factor  
    then Some DivisibleBy // 💡 Ou `Some ()`  
    else None  
  
let fizzBuzz = function  
    | DivisibleBy 3 &  
      DivisibleBy 5 → "FizzBuzz" // 💡 Ou `DivisibleBy 15`  
    | DivisibleBy 3 → "Fizz"  
    | DivisibleBy 5 → "Buzz"  
    | other → string other  
  
[1..15] ▷ List.map fizzBuzz  
// ["1"; "2"; "Fizz"; "4"; "Buzz"; "Fizz";  
//  "7"; "8"; "Fizz"; "Buzz"; "11";  
//  "Fizz"; "13"; "14"; "FizzBuzz"]
```

F#

# Fizz buzz avec active pattern : alternative

```
let isDivisibleBy factor number =  
    number % factor = 0  
  
let boolToOption b =  
    if b then Some () else None  
  
let (|Fizz|_|) number = number ▷ isDivisibleBy 3 ▷ boolToOption  
let (|Buzz|_|) number = number ▷ isDivisibleBy 5 ▷ boolToOption  
  
let fizzBuzz = function  
    | Fizz & Buzz → "FizzBuzz"  
    | Fizz → "Fizz"  
    | Buzz → "Buzz"  
    | other → string other
```

F#

→ Les 2 solutions se valent. C'est une question de style / de goût personnel.m

# Cas d'utilisation des actives patterns

1. Factoriser une guard (cf. *exercice précédent du fizz buzz*)
2. Wrapper une méthode de la BCL (cf. `(/Regex|_|)` et ci-dessous)
3. Améliorer l'expressivité, aider à comprendre la logique (cf. *après*)

```
let (|ParsedInt|UnparsableInt|) (input: string) =  
    match input with  
    | _ when fst (System.Int32.TryParse input) → ParsedInt(int input)  
    | _ → UnparsableInt  
  
let addOneOrZero = function  
    | ParsedInt i → i + 1  
    | UnparsableInt → 0  
  
let v1 = addOneOrZero "1" // 2  
let v2 = addOneOrZero "a" // 0
```

F#

# Expressivité grâce aux active patterns

```
type Movie = { Title: string; Director: string; Year: int; Studio: string }
```

F#

```
module Movie =
```

```
    let private boolToOption b =  
        if b then Some () else None
```

```
    let (|Director|_|) director movie =  
        movie.Director = director ▷ boolToOption
```

```
    let (|Studio|_|) studio movie =  
        movie.Studio = studio ▷ boolToOption
```

```
    let private matchYear comparator year movie =  
        (comparator movie.Year year) ▷ boolToOption
```

```
    let (|After|_|) = matchYear (>)
```

```
    let (|Before|_|) = matchYear (<)
```

```
    let (|In|_|) = matchYear (=)
```

# Expressivité grâce aux active patterns (2)

```
open Movie
```

```
F#
```

```
let ``Is anime rated 10/10`` = function  
| ((After 2001 & Before 2007) | In 2014) & Studio "Bones"  
| Director "Hayao Miyazaki" → true  
| _ → false
```

# Active pattern : citoyen de 1ère classe

Un active pattern  $\simeq$  fonction avec des métadonnées

→ Citoyen de 1ère classe :

```
// 1. Renvoyer un active pattern depuis une fonction
let (|Hayao_Miyazaki|_|) movie =
    (|Director|_|) "Hayao Miyazaki" movie

// 2. Prendre un active pattern en paramètre -- Un peu tricky
let firstItems (|Ok|_|) list =
    let rec loop values = function
        | Ok (item, rest) → loop (item :: values) rest
        | _ → List.rev values
    loop [] list

let (|Even|_|) = function
    | item :: rest when (item % 2) = 0 → Some (item, rest) | _ → None

let test = [0; 2; 4; 5; 6] ▷ firstItems (|Even|_|) // [0; 2; 4]
```

F#

# 4. ■ Le Récap'



# Récap' - Pattern matching

- Brique fondamentale de F#
- Combine "comparaison de structure de données" et "déconstruction"
- S'utilisent presque partout :
  - `match expression` et bloc `function`
  - bloc `try/with`
  - `let binding`, y.c. paramètre de fonction
- Peut s'abstraire en fonction `fold` associée à un type union



# Récap' - Patterns

Pattern	Exemple
Constant • Identifier • Wilcard	<code>1</code> , <code>Color.Red</code> • <code>Some 1</code> • <code>_</code>
<i>Collection</i> : Cons • List • Array	<code>head :: tail</code> • <code>[1; 2]</code>
<i>Product type</i> : Record • Tuple	<code>{ A = a }</code> • <code>a, b</code>
Type Test	<code>:? Subtype</code>
<i>Logique</i> : OR, AND	<code>1 \   2</code> , <code>P1 &amp; P2</code>
Variables • Alias	<code>head :: _</code> • <code>(0, 0) as origin</code>

+ Les guards `when` dans les match expressions

# Récap' - Active Patterns

- Extension du pattern matching
- Basés sur fonction + metadata → Citoyens de 1ère classe
- 4 types : total simple/multiple, partiel (simple), paramétré
- Un peu tricky à comprendre mais on s'habitue vite
- S'utilisent pour :
  - Ajouter de la sémantique sans recourir aux types union
  - Simplifier / factoriser des guards
  - Wrapper des méthodes de la BCL
  - Extraire un ensemble de données d'un objet
  - ...

# Compléments

 Match expressions

<https://fsharpforfunandprofit.com/posts/match-expression/>

 Domain modelling et pattern matching

<https://fsharpforfunandprofit.com/posts/roman-numerals/>

 Recursive types and folds (*6 articles*)

<https://fsharpforfunandprofit.com/series/recursive-types-and-folds/>

 A Deep Dive into Active Patterns

<https://www.youtube.com/watch?v=Q5KO-UDx5eA>

<https://github.com/pblasucci/DeepDiveAP>

# Exercices

Les exercices suivants sur <https://exercism.org/tracks/fsharp> peuvent se résoudre avec des active patterns :

- Collatz Conjecture (*easy*)
- Darts (*easy*)
- Queen Attack (*medium*)
- Robot Name (*medium*)

Merci 🙏

**SOAT**

→ Digitalize society



**SOAT.FR**