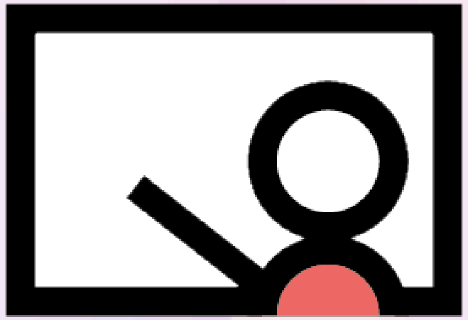
F♯ Training 📎

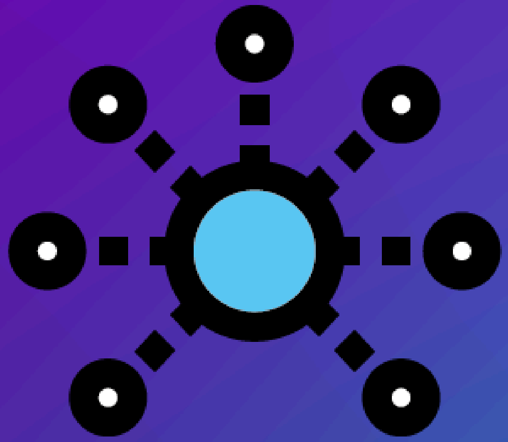
Pattern matching

2025 April

Table of contents

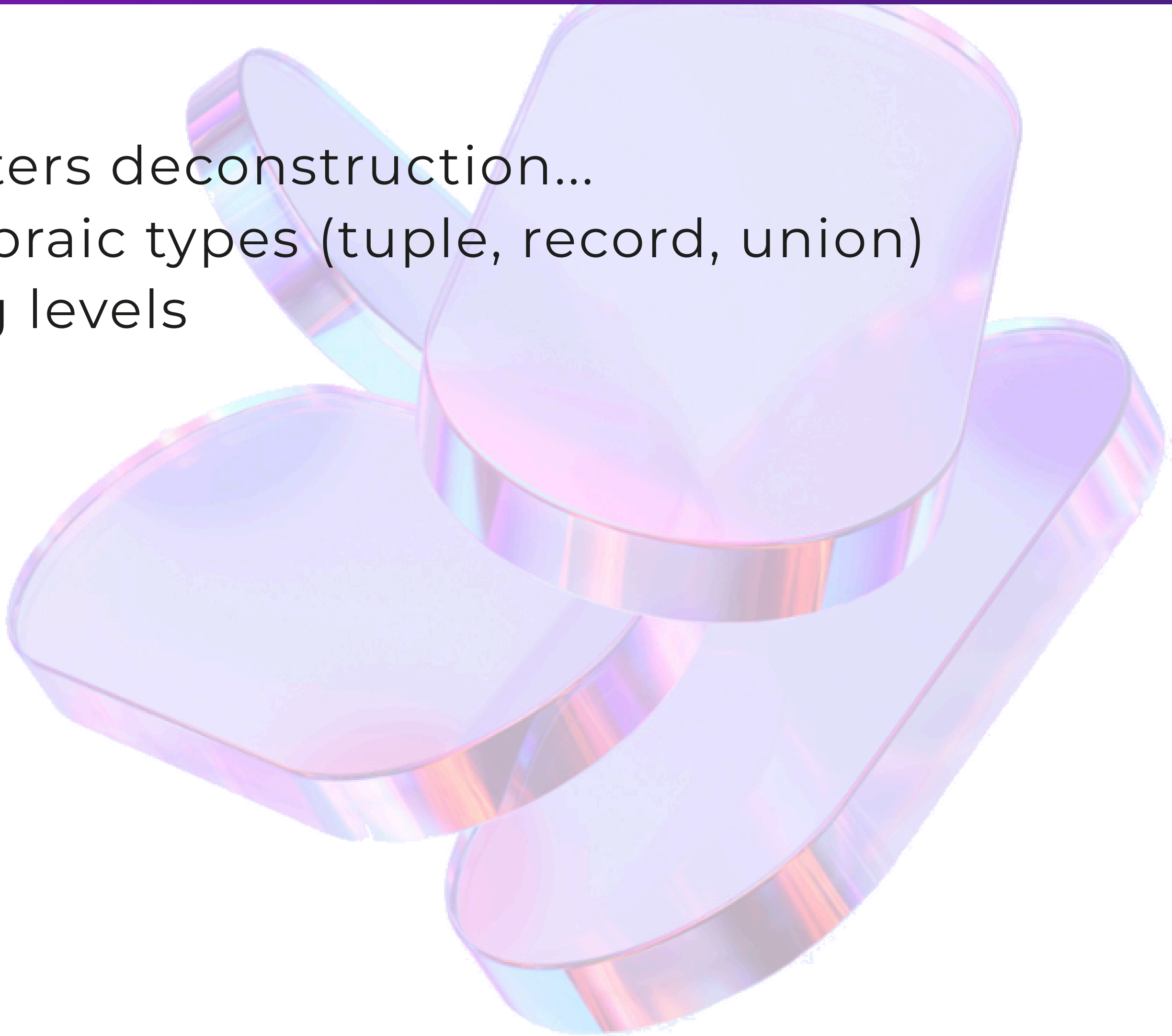


1. Patterns overview



Patterns

Used extensively in F♯
→ *match expression*, *let binding*, parameters deconstruction...
→ Very practical for manipulating F♯ algebraic types (tuple, record, union)
→ Composable: supports multiple nesting levels
→ Composed by logical AND/OR
→ Supports literals: `1.0`, `"test"`...



Wildcard Pattern

Represented by `_`, alone or combined with another *pattern*.

Always true
→ To be placed last in a *match expression*.

⚠️ Always seek 1st to handle all cases exhaustively/explicitly
When impossible, then use the `_`

```fsharp
match option with
| Some 1 -> ...
| _ -> ...              // ⚠️ Non exhaustive

match option with
| Some 1 -> ...
| Some _ | None -> ...  // 👌 More exhaustive
```

Constant Pattern

Detects constants, `null` and number literals, `char`, `string`, `enum`.

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

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

☝️ **Notes :**

➔ The `Three` pattern is also classified as an *Identifier Pattern* 📍

➔ For `null` matching, we also talk about *Null Pattern*.

Variable Pattern

Assigns the detected value to a "variable" for subsequent uses.

Example: `b` variable below

```fsharp
let isInt (s: string) =
    match System.Int32.TryParse(s) with
    | b, _ -> b
```

Variable Pattern (2)

⚠️ You cannot link to the same variable more than once.

```
let elementsAreEqualKo tuple =
    match tuple with
    | x, x -> true  // 💥 Error FS0038: x' is linked twice in this model
    | _, _ -> false
```

💡 **Solution:** use 2 variables then check their equality

```
// 1. Guard clause📍
let elementsAreEqualOk = function
    | x, y when x = y -> true
    | _, _ -> false

// 2. Deconstruction
let elementsAreEqualOk' (x, y) = x = y
```

Identifier Pattern

Detects *cases* of a union type and their possible contents

```fsharp
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"
```

Union case labelled fields

Several possibilities:
① "Anonymous" pattern of the complete tuple
② Pattern of a single field by its name → `Field = value`
③ Pattern of several fields by name → `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 ∅ {2*radius}"
```

Alias Pattern

`as` is used to name an element whose content is deconstructed

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

💡 Also works within functions to get back the parameter name:

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

let acceptMajorOnly ({ Age = age } as person) = // person: Person → Person option
    if age < 18 then None else Some person
```

OR / AND Patterns

Combine two patterns *(named `P1` and `P2` below)*.

- `P1 | P2` → P1 or P2. Ex: `Rectangle (0, _) | Rectangle (_, 0)`
- `P1 & P2` → P1 and P2. Used especially with *active patterns* 📍

💡 Use the same variable (`name` in the example below):

```
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+"
```

Parenthesized Pattern

Use of parentheses `()` to group patterns, to tackle precedence

```fsharp
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
```

☝️ **Note :** line ① would compile without doing `() :: tail`

Parenthesized Pattern (2)

⚠️ Parentheses complicate reading
💡 Try to do without when possible

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

Construction Patterns

Use type construction syntax to deconstruct a type

- ➔ Cons and List Patterns
- ➔ Array Pattern
- ➔ Tuple Pattern
- ➔ Record Pattern

Cons and List Patterns

≃ Inverses of the 2 ways to construct a list

Cons Pattern : `head :: tail` → decomposes a list *(with >= 1 element)* into:
- *Head*: 1st element
- *Tail*: another list with the remaining elements - can be empty

List Pattern : `[items]` → decompose a list into 0..N items
- `[]` : empty list
- `[x]` : list with 1 element set in the `x` variable
- `[x; y]` : list with 2 elements set in variables `x` and `y`
- `[_; _]`: list with 2 elements ignored

💡 `x :: []` ≡ `[x]`, `x :: y :: []` ≡ `[x; y]`...

Cons and List Patterns (2)

The default *match expression* combines the 2 patterns:
→ A list is either empty `[]`, or composed of an item and the rest: `head :: tail`

Recursive functions traversing a list use the `[]` pattern to stop recursion:

```
[<TailCall>]
let rec printList l =
    match l with
    | head :: tail →
        printf "%d " head
        printList tail
    | [] → printfn ""
```

Array Pattern

Syntax: `[| items |]` for 0..N items between `;`

```fsharp
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"
```

☝️ There is no pattern for sequences, as they are *"lazy "*.

Tuple Pattern

Syntax: `items` or `(items)` for 2..N items between `,`.

💡 Useful to match several values at the same time

```fsharp
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"
```

Record Pattern

Syntax: `{ Field1 = var1; … }`
→ Not required to specify all Record fields
→ In case of ambiguity, qualify the field: `Record.Field`

💡 Also works for function parameters:

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

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)

⚠️ **Reminder:** there is no pattern for anonymous *Records*!

```
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'  // 💥
```

Type Test Pattern

Syntax: `my-object :? sub-type` and returns a `bool`
→ ≃ `my-object is sub-type` in C♯

Usage: with a type hierarchy

```fsharp
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) "Windows cannot be registered!"
    | _ -> ()
```

Type Test Pattern - `try`/`with` block

This pattern is common in `try`/`with` blocks:

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

Type Test Pattern - Boxing

The *Type Test Pattern* only works with reference types.
→ For a value type or unknown type, it must be boxed.

```
let isIntKo = function :? int → true | _ → false
//                     ~~~~~~~
// 💥 Error FS0008: This runtime coercion or type test from type 'a to int
//    involves an indeterminate type based on information prior to this program point.

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

2. *Match Expression*

Match expression

Similar to a `switch` expression in C♯ 8.0 but more powerful thanks to patterns

Syntax:

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

Returns the result of the 1st branch whose pattern "matches" `test-expression`

☝ **Note:** all branches must return the same type!

Match expression - Exhaustivity

A C# `switch` must always define a default case.
Otherwise: compile warning, 💥 `MatchFailureException` at runtime

Not necessary in a F# *match expression* if branches cover all cases because the compiler checks for completeness and "dead" branches

```fsharp
let fn x =
    match x with
    | Some true  → "ok"
    | Some false → "ko"
    | None       → ""
    | _          → "?"
//    ~  ⚠️ Warning FS0026: his rule will never be matched
```

Match expression - Exhaustivity (2)

☝️ **Tip:** the more branches are exhaustive, the more code is explicit and safe

Example: checking all the cases of a union type allows you to manage the addition of a case by a warning at compile time:

```
Warning FS0025: Special criteria incomplete in this expression
```

- ➔ Detection of accidental addition
- ➔ Identification of the code to change to handle the new case

Match expression - Guard

Syntax: `pattern1 when condition`
Usage: to refine a pattern, using constraints on variables

```fsharp
let classifyBetween low top value =
    match value with
    | x when x < low -> "Inf"
    | 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"
```

💡 The *guard* is only evaluated if the pattern is satisfied.

Match expression - Guard *vs* OR Pattern

The OR pattern has a higher *precedence/priority* than the *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  // 👈 The guard is covering the 2 previous patterns
    | _ -> false

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

Match function

Syntax:

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

Equivalent to a lambda taking an implicit parameter which is "matched":

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

Match function - Interest

1. In pipeline

```
value
|> is123
|> function
    | true  -> "ok"
    | false -> "ko"
```

2. Terser function

```
let is123 = function
    | 1 | 2 | 3 -> true
    | _ -> false
```

Match function - Limitations

⚠️ Implicit parameter => can make the code more difficult to understand!

Example: function declared with other explicit parameters
→ The number of parameters and their order can be wrong:

```fsharp
let classifyBetween low high = function  // 👈 3 parameters : `low`, `high`, and another one implicit
    | 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"
```

Exhaustivity in OOP

The equivalent of the pattern matching exhaustivity in FP is
... the Visitor design pattern in OOP

> Visitor is a behavioral design pattern that lets you **separate algorithms from the objects** on which they operate.

→ It's FP in OOP, much very convoluted: see *double-dispatch technique*

`fold` function 🚀

Function associated with a union type and hiding the *matching* logic
Takes N+1 parameters for a union type with N *cases*

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

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

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

`fold` function: usage

```
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 fahrenheintToCelsius x = (x - DeltaC2F) / FactorC2F  // float<F> → float<C>

    let toggleUnit temperature =
        temperature ▷ fold
            (celsiusToFahrenheint >> Fahrenheint)
            (fahrenheintToCelsius >> Celsius)

let t1 = Celsius 100.0<C>
let t2 = t1 ▷ Temperature.toggleUnit  // Fahrenheint 212.0
```

`fold` function: interest

`fold` hides the implementation details of the type

For example, we could add a `Kelvin` *case* and only impact `fold`,
not the functions that call it, such as `toggleUnit` in the previous example

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

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

// Code continued on next slide...
```

`fold` function: interest (2)

```fsharp
// ...
module Temperature =
    let fold mapCelsius mapFahrenheint temperature : 'T =
        match temperature with
        | Celsius x      → mapCelsius x       // mapCelsius: float<C> → 'T
        | Fahrenheint x → mapFahrenheint x  // mapFahrenheint: float<F> → 'T
        | Kelvin x       → mapCelsius (x * 1.0<C/K> + 273.15<C>)  // 🌟

Kelvin 273.15<K>
▷ Temperature.toggleUnit
// Celsius 0.0<C>
```

3. *Active Patterns*

Pattern Matching Limits

Limited number of patterns

Impossibility of factoring the action of patterns with their own guard
→ `Pattern1 when Guard1 | Pattern2 when Guard2 -> do` 💥
→ `Pattern1 when Guard1 -> do | Pattern2 when Guard2 -> do` 😕

Patterns are not 1st class citizens
Ex: a function can't return a pattern
→ Just a kind of syntactic sugar

Patterns interact badly with an OOP style

Origin of *Active Patterns*

> 🔗 [*Extensible pattern matching via a lightweight language extension*](#)
> ℹ️ 2007 publication by Don Syme, Gregory Neverov, James Margetson

Integrated into F♯ 2.0 (2010)

💡 **Ideas**

- ➔ Enable *pattern matching* on other data structures
- ➔ Make these new patterns 1st class citizens

Active Patterns - Syntax

General syntax : `let (|Cases|) [arguments] valueToMatch = expression`

1. **Function** with a special name defined in a "banana" `(| … |)`
2. Set of 1..N **cases** in which to store `valueToMatch` parameter

💡 Kind of *factory* function of an "anonymous" **union** type, defined *inline*

Active Patterns - Types

There are 4 types of active patterns:

Name	Cases	Exhaustive	Parameters
`1. Simple Total`	1	✅ Yes	❔ 0+
`2. Multiple Total`	2+	✅ Yes	❌ 0
`3. Partial`	1	❌ No	❌ 0
`4. Parametric`	1	❌ No	✅ 1+

Simple total active pattern

A.k.a Single-case Total Pattern

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

Usage: on-site value adjustment

```
/// Ensure the given string is never null
let (|NotNullOrEmpty|) (s: string) = // string → string
    if s ▷ isNull then System.String.Empty else s

// Usages:
let (NotNullOrEmpty a) = "abc" // val a: string = "abc"
let (NotNullOrEmpty b) = null  // val b: string = ""
```

Simple total active pattern (2)

Can accept **parameters** → ⚠️ usually more difficult to understand

```fsharp
/// Get the value in the given option if there is some, otherwise the specified default value
let (|Default|) defaultValue option = option |> Option.defaultValue defaultValue
//              'T     -> 'T option -> 'T

// Usages:
let (Default "unknown" john) = Some "John"  // val john: string = "John"
let (Default 0 count) = None                // val count: int = 0

// Template function
let (|ValueOrUnknown|) = (|Default|) "unknown" // string option -> string

let (ValueOrUnknown person) = None  // val person: string = "unknown"
```

Simple total active pattern (3)

Another example: extracting the polar form of a complex number

```
/// Extracts the polar form (Magnitude, Phase) of the given complex number
let (|Polar|) (x: System.Numerics.Complex) =
    x.Magnitude, x.Phase

/// Multiply the 2 complex numbers by adding their phases and multiplying their magnitudes
let multiply (Polar(m1, p1)) (Polar(m2, p2)) =  // Complex → Complex → Complex
    System.Numerics.Complex.FromPolarCoordinates(magnitude = m1 * m2, phase = p1 + p2)

// Without the active pattern: we need to add type annotations
let multiply' (x: System.Numerics.Complex) (y: System.Numerics.Complex) =
    System.Numerics.Complex.FromPolarCoordinates(x.Magnitude * y.Magnitude, x.Phase + y.Phase)
```

Active pattern total multiple

A.k.a Multiple-case Total Pattern

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

☝ No parameters ❗

```fsharp
// Using an ad-hoc union type
type Parity = Even of int | Odd of int with
    static member Of(x) =
        if x % 2 = 0 then Even x else Odd x

let hasSquare square value =
    match Parity.Of(square), Parity.Of(value) with
    | Even sq, Even v
    | Odd  sq, Odd  v when sq = v*v -> true
    | _ -> false
```

```fsharp
// Using a total active pattern
let (|Even|Odd|) x =  // int -> Choice<int, in
    if x % 2 = 0 then Even x else Odd x

let hasSquare' square value =
    match square, value with
    | Even sq, Even v
    | Odd  sq, Odd  v when sq = v*v -> true
    | _ -> false
```

Partial active pattern

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

→ Returns the type `'T option` if *Case* includes data, otherwise `unit option`

→ Pattern matching is non-exhaustive → a default case is required

```
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"
```

Parametric partial active pattern

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

Example 1: leap year

→ Year multiple of 4 but not 100 except 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
```

Parametric partial active pattern (2)

Example 2: Regular expression

```
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
```

💡 Usages seen with the next example…

Parametric partial active pattern (3)

Example 3: Hexadecimal color

```fsharp
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
    // 👇 Uses the previous active pattern
    // 💡 The Regex searches for 3 groups of 2 chars being a number or a letter 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"
```

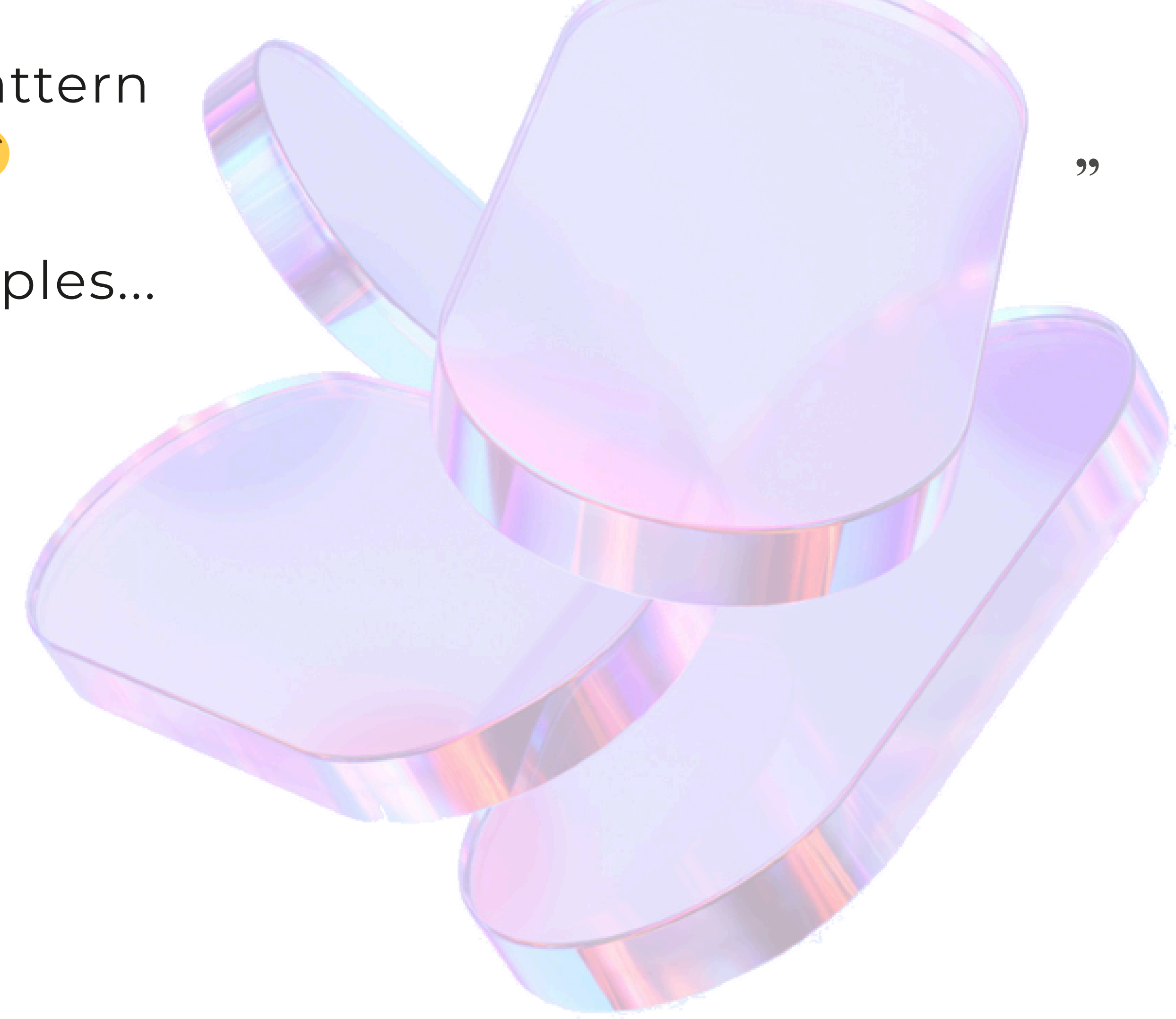
Active patterns recap

```
Active pattern │ Syntax                           │ Signature
───────────────┼──────────────────────────────────┼──────────────────────────────────────────────────
Total multiple │ let (|Case1|..|CaseN|)         x │                    'T → Choice<'U1, .., 'Un>
... parametric │ let (|Case1|..|CaseN|) p1..pp x │ 'P1 → .. → 'Pp → 'T → Choice<'U1, .., 'Un>
Total simple   │ let (|Case|)                   x │                    'T → 'U
Partial simple │ let (|Case|_|)                 x │                    'T → 'U option
... parametric │ let (|Case|_|)         p1..pp x │ 'P1 → .. → 'Pp → 'T → 'U option
```

Understanding an active pattern

> “ Understanding how to use an active pattern can be a real **intellectual challenge**! 😵 ”

👉 Explanations using the previous examples...



Understanding a total active pattern

≃ *factory* function of an "anonymous" union type

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

let (Cartesian(r, i)) = System.Numerics.Complex(1.0, 2.0)
// val r: float = 1.0
// val i: float = 2.0

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

let printParity = function
    | Even as n -> printfn $"%i{n} is even"
    | Odd  as n -> printfn $"%i{n} is odd"

printParity 1;;  // 1 is odd
printParity 10;; // 10 is even
```

Understanding a partial active pattern

☝️ Distinguish parameters *(input)* from data *(output)*

Examine the active pattern signature: `[ … params →] value → 'U option`

- ➔ **N-1 parameters:** active pattern parameters
- ➔ **Last parameter:** `value` to match
- ➔ **Return type:** `'U option` → data of type `'U`
  - ➔ when `unit option` → no data

Understanding a partial active pattern (2)

Examples

1. `let (|Integer|_|) (s: string) : int option`
   Usage `match s with Integer i` → `i: int` is the output data
2. `let (|DivisibleBy|_|) (factor: int) (x: int) : unit option`
   Usage `match year with DivisibleBy 400` → `400` is the `factor` parameter
3. `let (|Regexp|_|) (pattern: string) (value: string) : string list option`
   Usage `match s with Regexp "#([0-9 ... )" [ r; g; b ]`
   → `"#([0-9 ... )"` is the `pattern` parameter
   → `[ r; g; b ]` is the output data · It's a nested pattern: a list of 3 strings

Exercise: fizz buzz with active pattern

Rewrite this fizz buzz using an active pattern `DivisibleBy`.

```fsharp
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"]
```

Fizz buzz with active pattern: solution

```fsharp
let isDivisibleBy factor number =
    number % factor = 0

let (|DivisibleBy|_|) factor number =
    if number |> isDivisibleBy factor then Some () else None
    // 👆 In F# 9, just `number |> isDivisibleBy factor` is enough 👍

let fizzBuzz = function
    | DivisibleBy 15 -> "FizzBuzz"
    | 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"]
```

Fizz buzz with active pattern: alternative

```fsharp
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
```

→ The 2 solutions are equal. It's a matter of style / personal taste.

→ In F♯ 9, no need to do `|> boolToOption`.

Active patterns use cases

1. Factor a guard *(see previous fizz buzz exercise)*
2. Wrapping a BCL method *(see `(|Regexp|_|)` and below).*
3. Improve expressiveness, help to understand logic *(see below)*

```fsharp
[<RequireQualifiedAccess>]
module String =
    let (|Int|_|) (input: string) = // string -> int option
        match System.Int32.TryParse(input) with
        | true, i  -> Some i
        | false, _ -> None

let addOneOrZero = function
    | String.Int i -> i + 1
    | _ -> 0

let v1 = addOneOrZero "1"  // 2
let v2 = addOneOrZero "a"  // 0
```

Expressiveness with active patterns

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

module Movie =
    let inline private satisfy ([<InlineIfLambda>] predicate) (movie: Movie) =
        match predicate movie with
        | true -> Some ()
        | false -> None

    let (|Director|_|) director = satisfy (fun movie -> movie.Director = director)
    let (|Studio|_|) studio = satisfy (fun movie -> movie.Studio = studio)
    let (|In|_|) year = satisfy (fun movie -> movie.Year = year)
    let (|Between|_|) min max = satisfy (fun { Year = year } -> year >= min && year <= max)

// ...
```

Expressiveness with active patterns (2)

```fsharp
// ...

open Movie

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

let topAnimes =
    [ { Title = "Cowboy Bebop"; Director = "Shinichirō Watanabe"; Year = 2001; Studio = "Bones" }
      { Title = "Princess Mononoke"; Director = "Hayao Miyazaki"; Year = 1997; Studio = "Ghibli" } ]
    |> List.filter ``Is anime rated 10/10``
```

Active pattern: 1st class citizen

An active pattern ≃ function with metadata → 1st class citizen in F#

```fsharp
// 1. Return an active pattern from a function
let (|Hayao_Miyazaki|_|) movie =
    (|Director|_|) "Hayao Miyazaki" movie

// 2. Take an active pattern as parameter -- A bit 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]
```

4. Wrap up

Wrap up Pattern matching

- F♯ core building block
- Combines "data structure matching" and "deconstruction"
- Used almost everywhere:
  - `match` and `function` expressions
  - `try/with` block
  - `let` binding, including function parameter
- Can be abstracted into a `fold` function associated with a union type

Wrap up Patterns

Pattern	Example
Constant · Identifier · Wilcard	`1`, `Color.Red` · `Some 1` · `_`
Collection : Cons · List · Array	`head :: tail` · `[1; 2]`
Product type : Record · Tuple	`{ A = a }` · `a, b`
Type Test	`:? Subtype`
Logical : OR, AND	`1 | 2`, `P1 & P2`
Variables · Alias	`head :: _` · `(0, 0) as origin`

+ The `when` guards in `match` expressions

Wrap up Active Patterns

- ➔ Extending pattern matching
- ➔ Based on function + metadata → 1st-class citizens
- ➔ 4 types: total simple/multiple, partial (simple), parametric
- ➔ At 1st little tricky to understand, but we get used to it quickly
- ➔ Use for:
  - ➔ Add semantics without relying on union types
  - ➔ Simplify / factorize guards
  - ➔ Wrapping BCL methods
  - ➔ Extract a data set from an object
  - ➔ …

Addendum

📜 Match expressions
https://fsharpforfunandprofit.com/posts/match-expression/

📜 Domain modelling and 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

Exercises

The following exercises on https://exercism.org/tracks/fsharp can be solved with active patterns:

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

Thanks 🙏