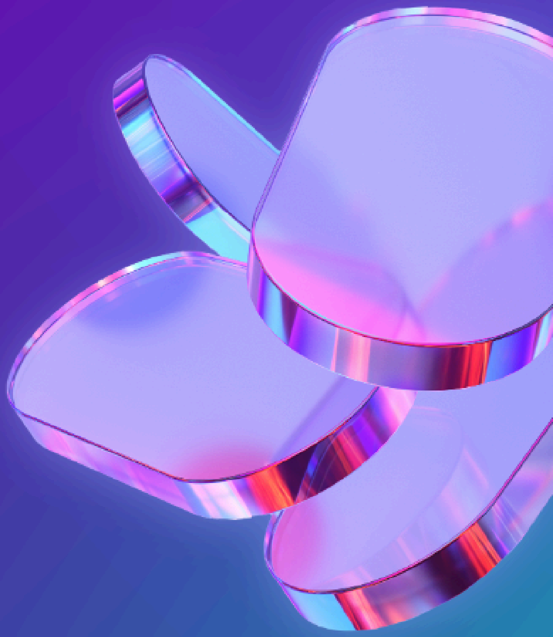
F♯ Training

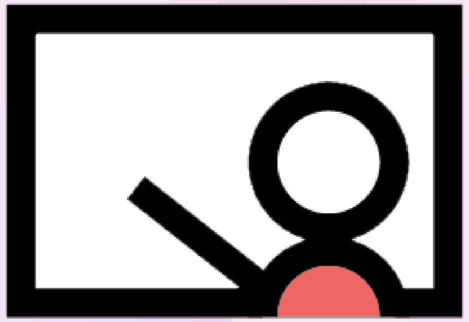
Functions

2025 April



d-edge

Table of contents



1. Function signature

Problems with `void` in C♯

`void` forces you to be specific = 2 times more work 😠

- ➔ 2 types of delegates: `Action` vs `Func<T>`
- ➔ 2 types of tasks: `Task` vs `Task<T>`

Example:

```
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);
}
```

From `void` keyword to `Void` type

☝️ The problem with `void` is that it's neither a **type** nor a **value**.

💡 If we had a `Void` type, a *Singleton* of type :

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

    private Void() {}
}
```

From `void` keyword to `Void` type (2)

The following *helpers* can be defined to convert to `Void` :

```csharp
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;
    }
}
```

Simplifying ITelemetry

We can write a default implementation (C♯ 8) for 2 of the 4 methods:

```
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);
}
```

In F♯, `Void` is called `Unit`.

In F♯, no `void` function but functions with return type `Unit` / `unit`.

`unit` has a single instance (hence its name), noted `()`.
→ Used as the last expression of a `void` function:

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

Parameterless functions

`unit` is also used to model parameter-free functions:

```fsharp
let oneParam arg = ...
let noParam () = ... // 👈 With
let noParam2() = ... // 👈 or without space
```

💡 Advantages of `()` notation: looks like a C♯ function.

⚠️ **Warning:** it's easy to forget the `()`!

- ➔ Omission in the declaration → simple value rather than function
- ➔ forget in the call → alias the function without executing it

Function `ignore`

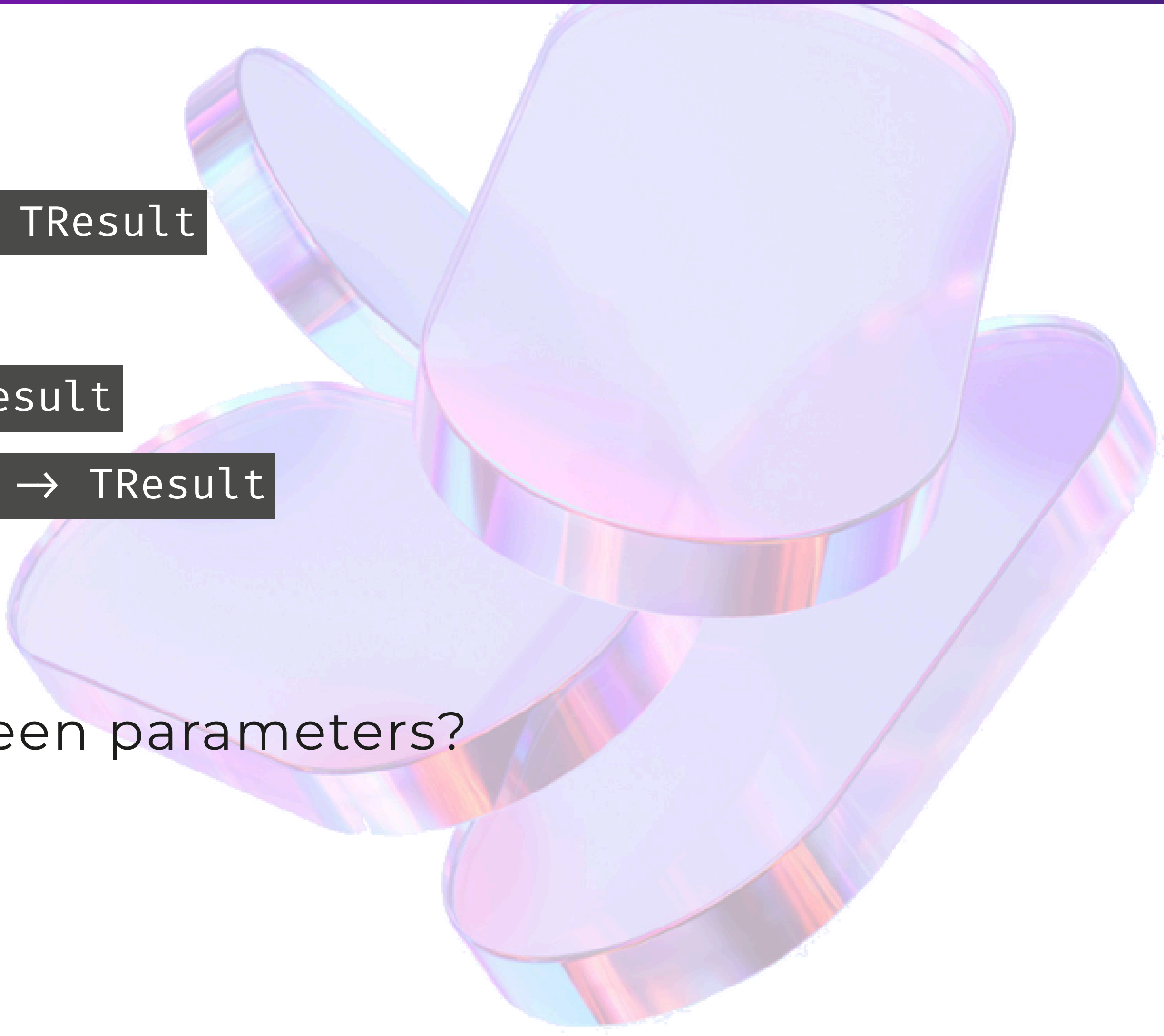
In F♯, everything is expression, but you can insert `unit` expressions, for example a `printf` before returning the value.

Problem: calling a `save` function to save in base, but it returns the `true` or `false` value you want to ignore.

Solution: use the `ignore` signature function `'a → unit`.
→ Whatever the value supplied as a parameter, it ignores it and returns `()`.

```fsharp
let save entity = true

let a =
    save "bonjour" // ⚠️ Warning FS0020: The result of this expression has the type 'bool' and is implicitly
    save "bonjour" ▷ ignore // 👌
    "ok"
```

Signature of a function in F♯



Arrow notation:

- ➔ Function with 0 parameters: `unit → TResult`
- ➔ 1-parameter function: `T → TResult`
- ➔ 2-parameter function: `T1 → T2 → TResult`
- ➔ 3-parameter function: `T1 → T2 → T3 → TResult`

❓ **Quiz** ❓

- ➔ Why several `→` rather than `,` between parameters?
- ➔ What is the underlying concept?

Currying

F♯ function syntax: parameters separated by spaces
→ Indicates that functions are curried
→ Hence the `→` in the signature between parameters

```
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, explicitly curried :
let fn x = fun y → (x, y) // 1. With a lambda
let fn x =                  // 2. With a sub-function
    let fn' y = (x, y)      // N.B. `x` comes from the enclosing scope
    fn'
```

Currying - .NET Compilation

☝️ Curried function compiled as a method with tuplified parameters
→ Viewed as normal method when consumed in C♯

Example : F♯ then equivalent C♯ *(simplified from SharpLab)* :

```fsharp
module A =
    let add x y = x + y
    let value = 2 |> add 1
```

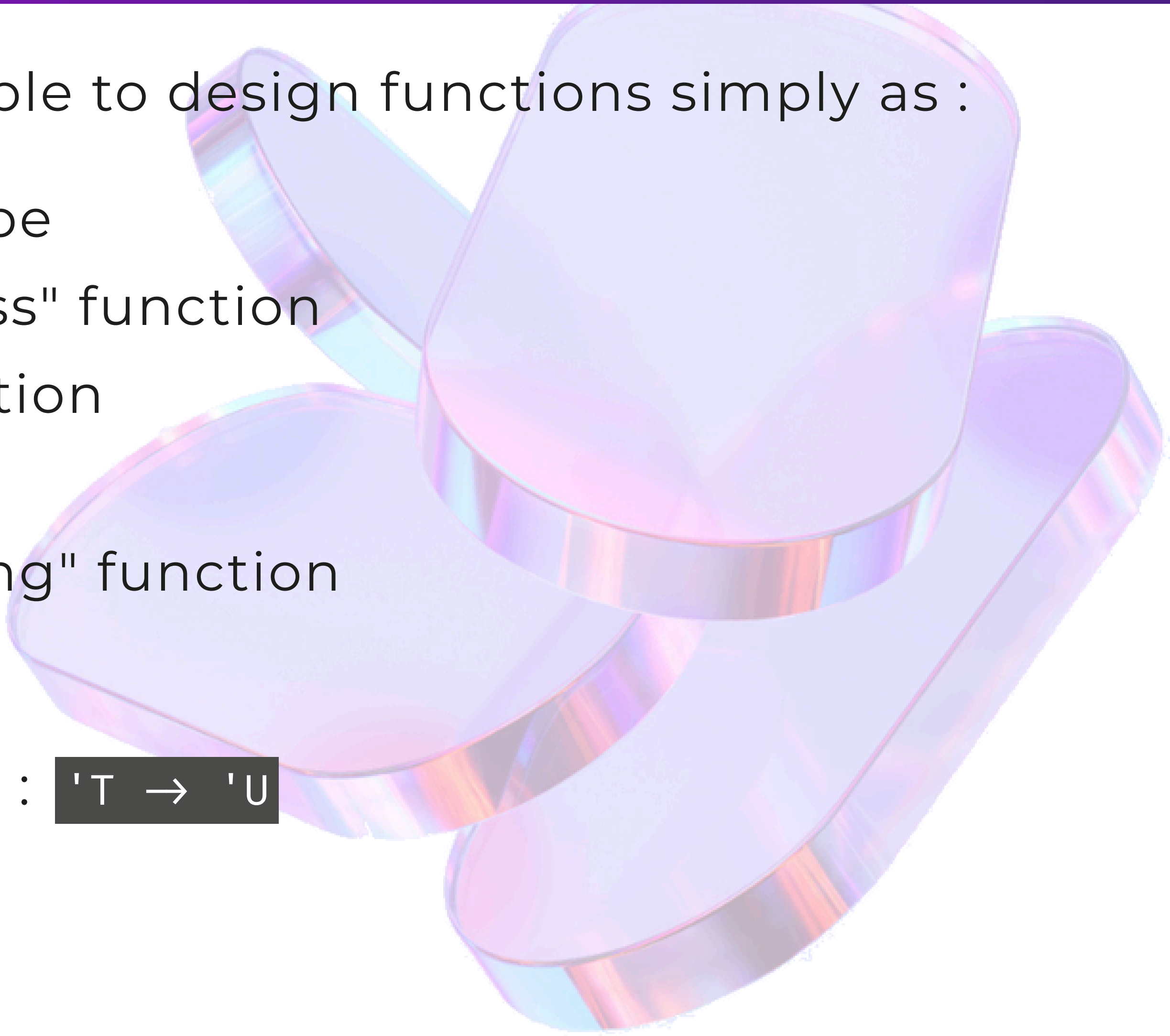
```csharp
public static class A
{
    public static int add(int x, int y) => x + y;
    public static int value => 3;
}
```

Unified function design

The `unit` type and currying make it possible to design functions simply as :

- ➔ **Takes a single parameter** of any type
  - ➔ including `unit` for a "parameterless" function
  - ➔ including another *(callback)* function
- ➔ Returns a single value of any type
  - ➔ including `unit` for a "return nothing" function
  - ➔ including another function

👉 **Universal signature** of a function in F♯ : `'T → 'U`



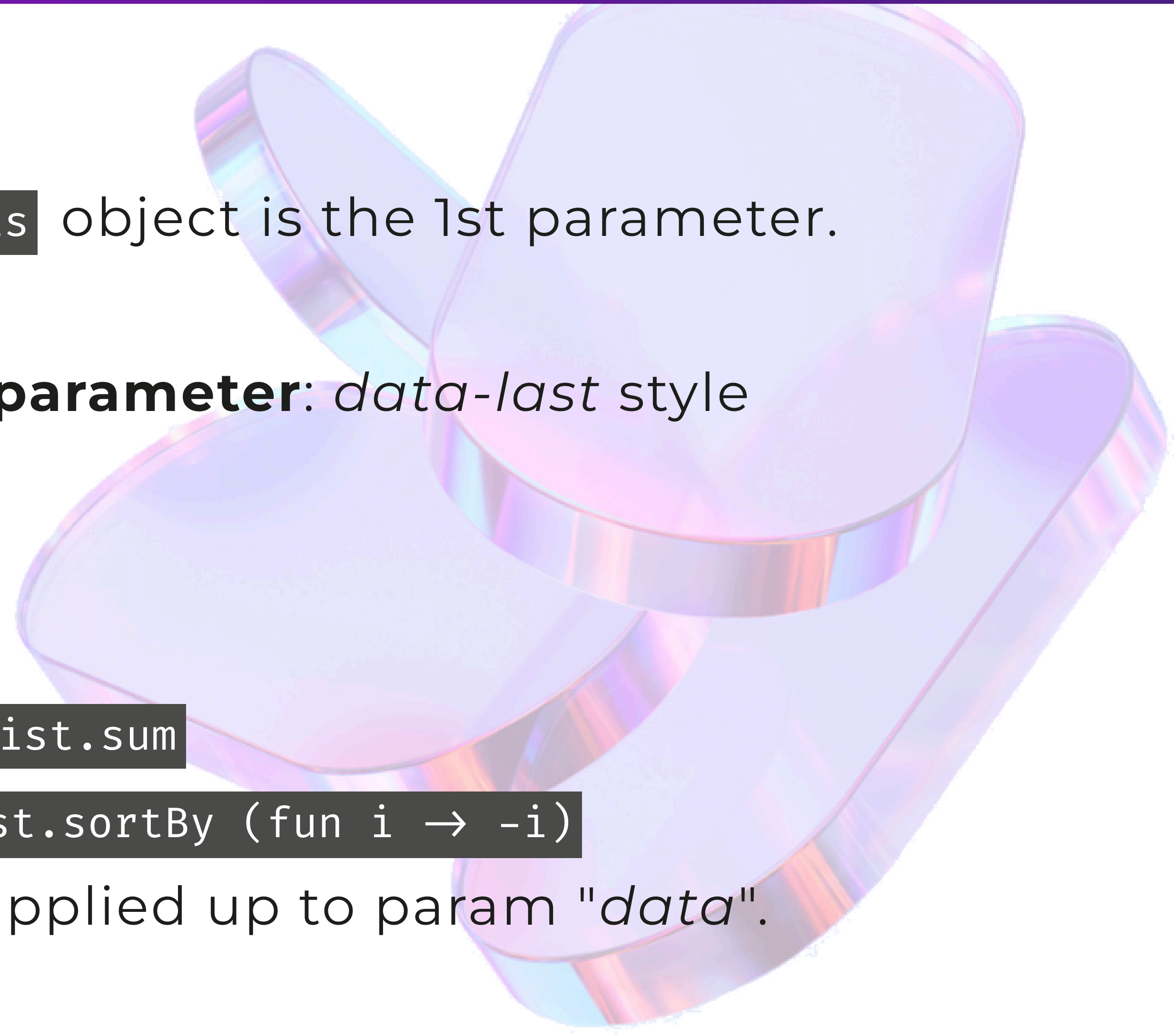
Order of parameters

Not the same order between C♯ and F♯

- ➔ In the C♯ extension method, the `this` object is the 1st parameter.
  - ➔ Ex: `items.Select(x => x)`
- ➔ In F♯, "the object" is rather the **last parameter**: *data-last* style
  - ➔ Ex: `List.map (fun x -> x) items`

Style *data-last* favors:

- ➔ Pipeline: `items |> List.map square |> List.sum`
- ➔ Partial application: `let sortDesc = List.sortBy (fun i -> -i)`
- ➔ Composition of functions partially applied up to param "*data*".
  - ➔ `(List.map square) >> List.sum`



Order of parameters (2)

⚠️ Friction with BCL .NET as *data-first* is more appropriate

☝️ Solution: wrap in a function with params in a nice F♯ order

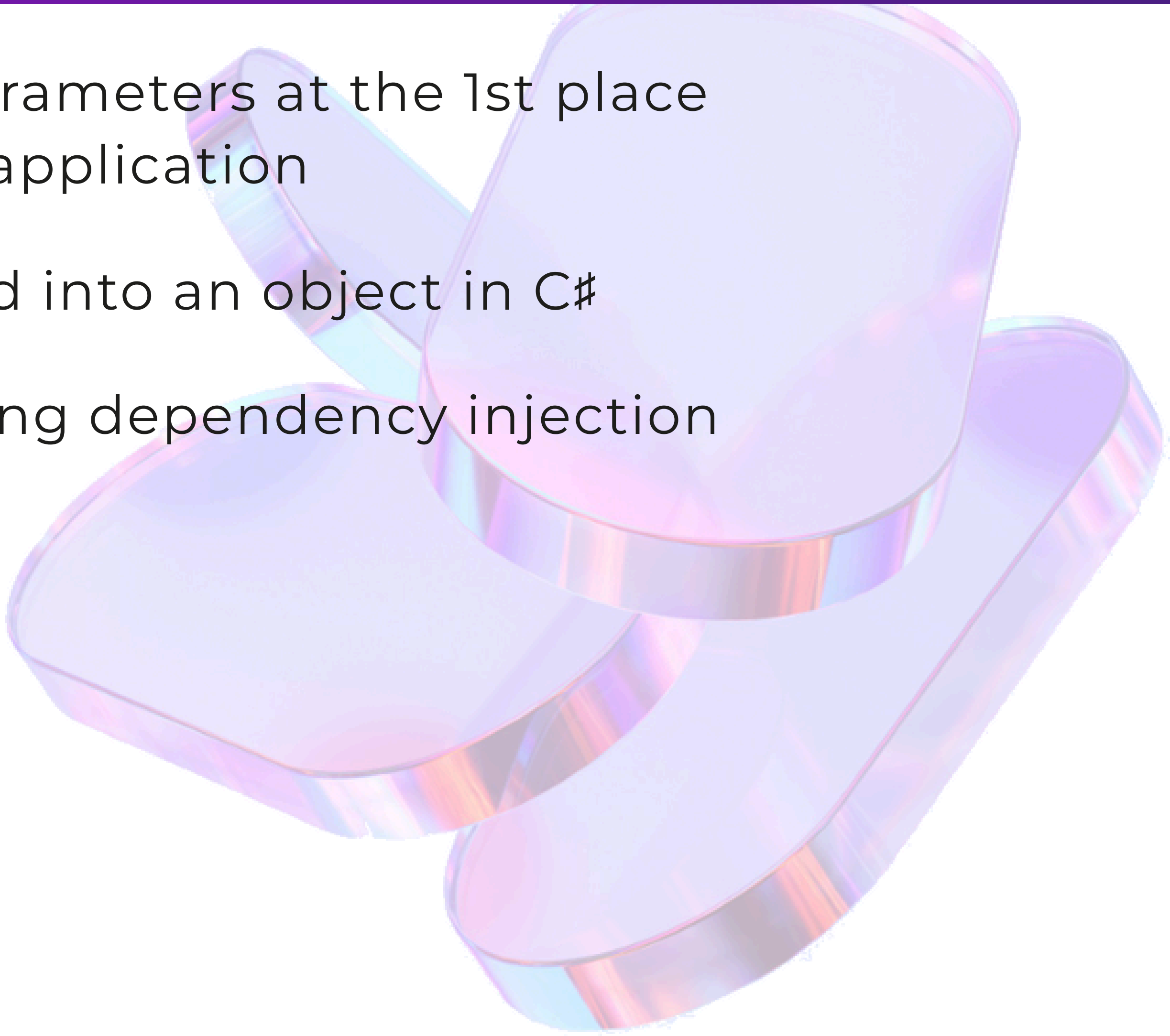
```fsharp
let startsWith (prefix: string) (s: string) =
    s.StartsWith(prefix)
```

Order of parameters (3)

In the same way, place the most static parameters at the 1st place
= those likely to be predefined by partial application

Ex: "dependencies" that would be injected into an object in C♯

👉 Partial application = means of simulating dependency injection

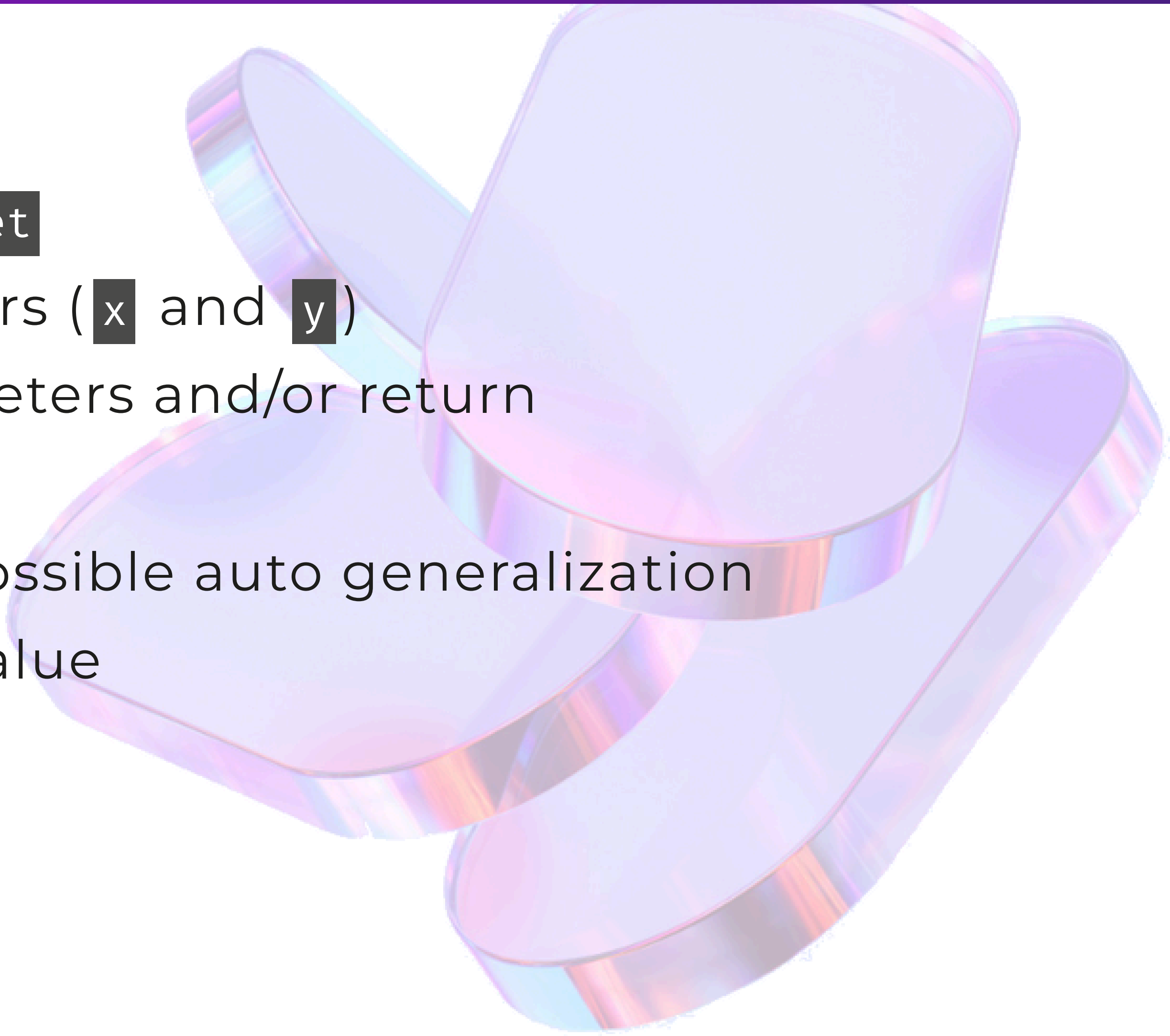


2. Functions Syntax

Binding a function

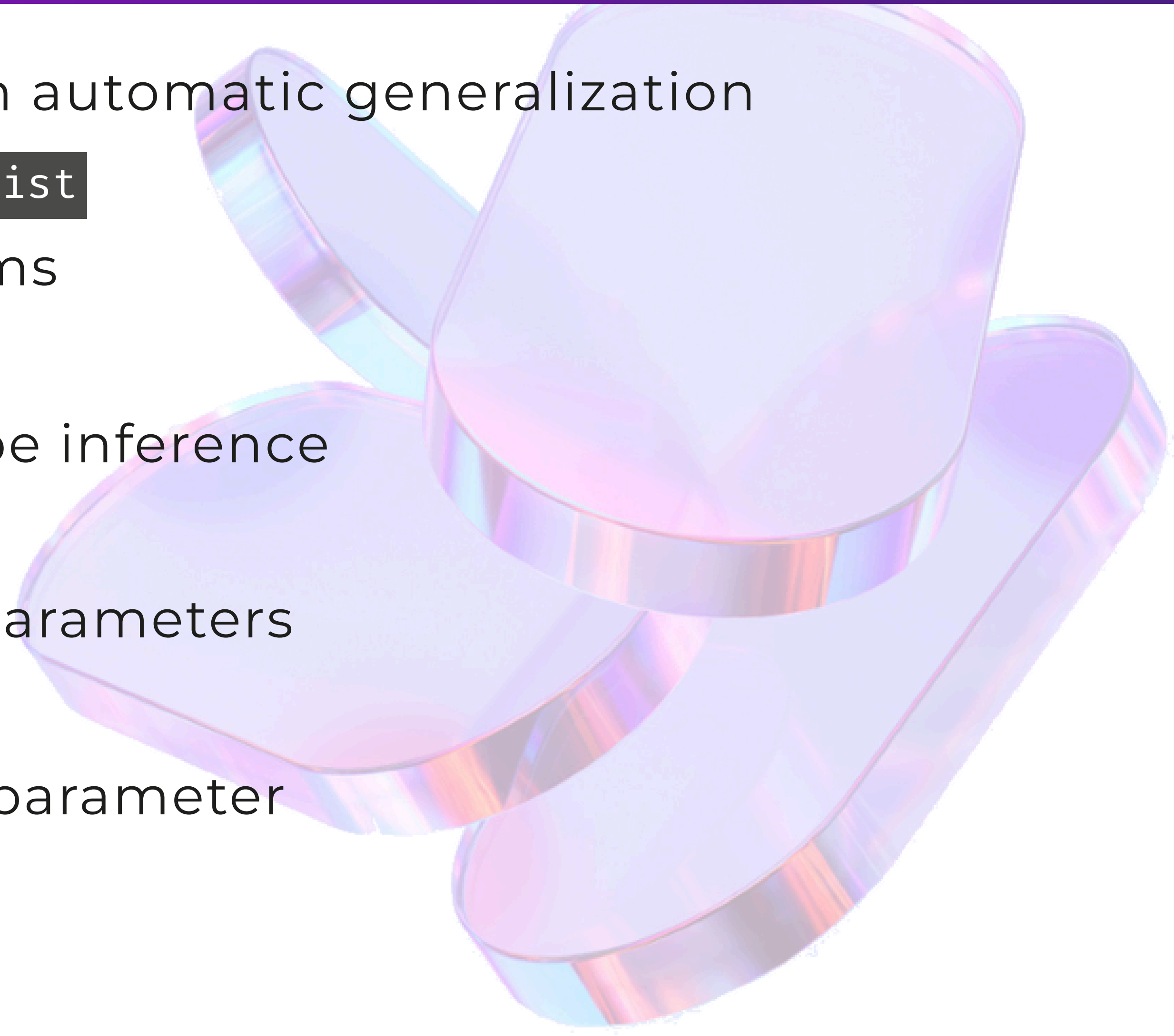
```
let f x y = x + y + 1
```

- ➔ Binding performed with keyword `let`
- ➔ Binds both name (`f`) and parameters (`x` and `y`)
- ➔ Optional type annotation for parameters and/or return
  - ➔ `let f (x: int) (y: int) : int = ...`
  - ➔ Otherwise, type inference, with possible auto generalization
- ➔ Last expression → function return value
- ➔ Can contain nested functions



Generic function

- In many cases, inference works with automatic generalization
  - `let listOf x = [x]` → `(x: 'a) → 'a list`
- Explicit annotation of generic params
  - `let f (x: 'a) = ...`
- Explicit annotation with generic type inference
  - `let f (list: list<_>) = ...`
- Full explicit annotation of generic parameters
  - `let f<'a> (x: 'a) = ...`
  - Pros: callers can specify the type parameter



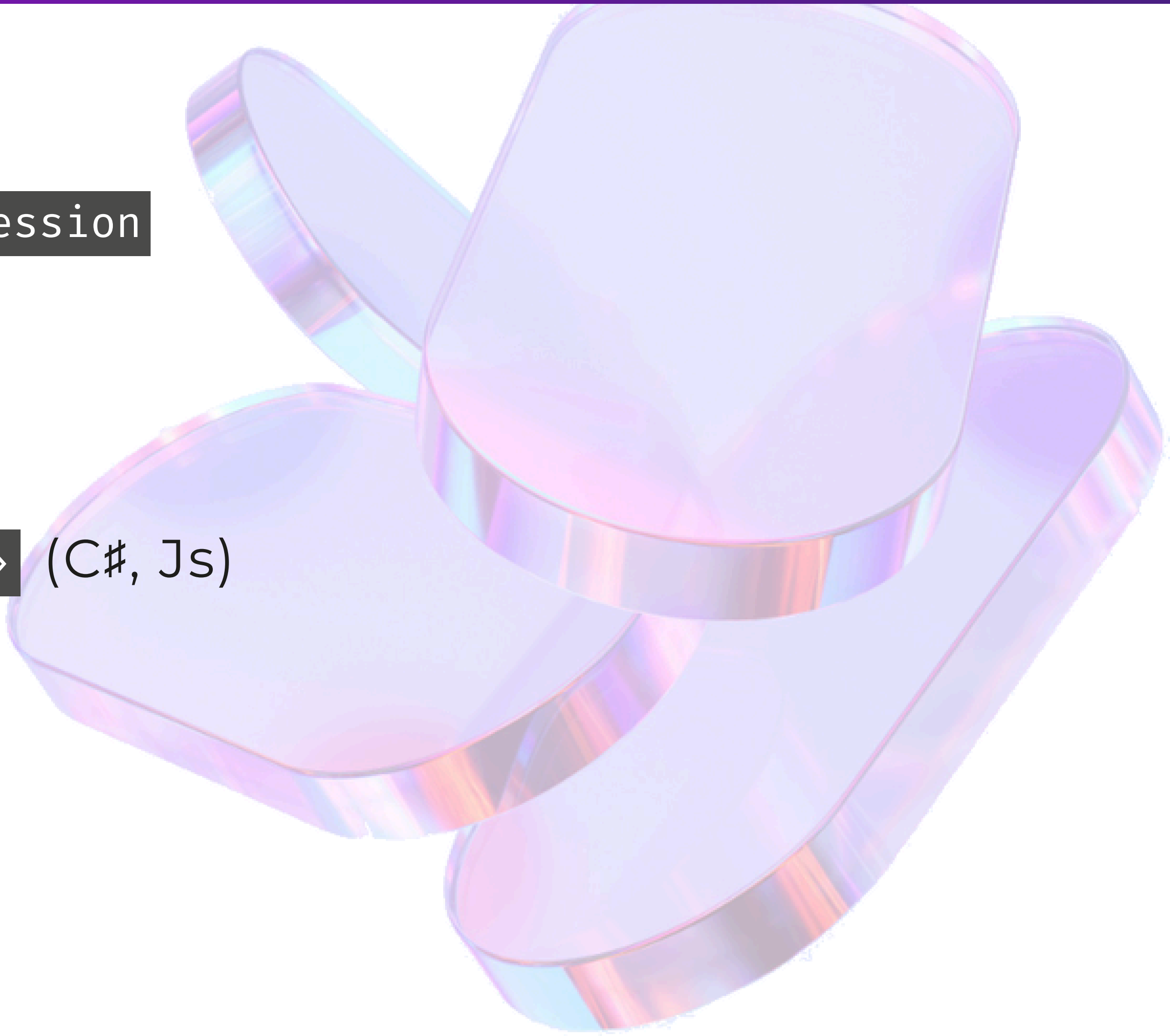
Anonymous function / Lambda

Expression defining a function

Syntax: `fun parameter1 parameter2 etc → expression`

☝️ **Note:**

- ➔ Keyword `fun` mandatory
- ➔ Thin arrow `→` (Java) ≠ Bold arrow `⇒` (C♯, Js)



Anonymous functions - Some use cases

1. As an argument to a *high-order function*

➔ To avoid having to define a named function

➔ Recommended for a short function, to keep it readable

```
[1..10] |> List.map (fun i -> i + 1) // 👈 () around the lambda

// Versus a function named
let add1 i = i + 1
[1..10] |> List.map add1
```

⚠️ Useless lambda: `List.map (fun x -> f x)` ≡ `List.map f`

2. In *let binding* with inference

- ➔ To make explicit when the function returns a function
- ➔ A kind of manual currying
- ➔ Use sparingly

```
let add x y = x + y // Normal version, automatically curried
let add' x = fun y → x + y // Same with a lambda sub
let add'' = fun x → (fun y → x + y) // Same, totally lambda-ized
```

3. let binding with type annotations

- ➔ Pre-defined function signature in the form of a type
- ➔ Type "function" is used like a C♯ `interface`
  - ➔ To force implementation to follow signature
  - ➔ Ex: *Domain modelling made functional* by Scott Wlaschin

```
type Add = int → int → int

let add: Add = fun x y → x + y // 👈 Final signature with named param : (x: int) → (y: int) → int
```

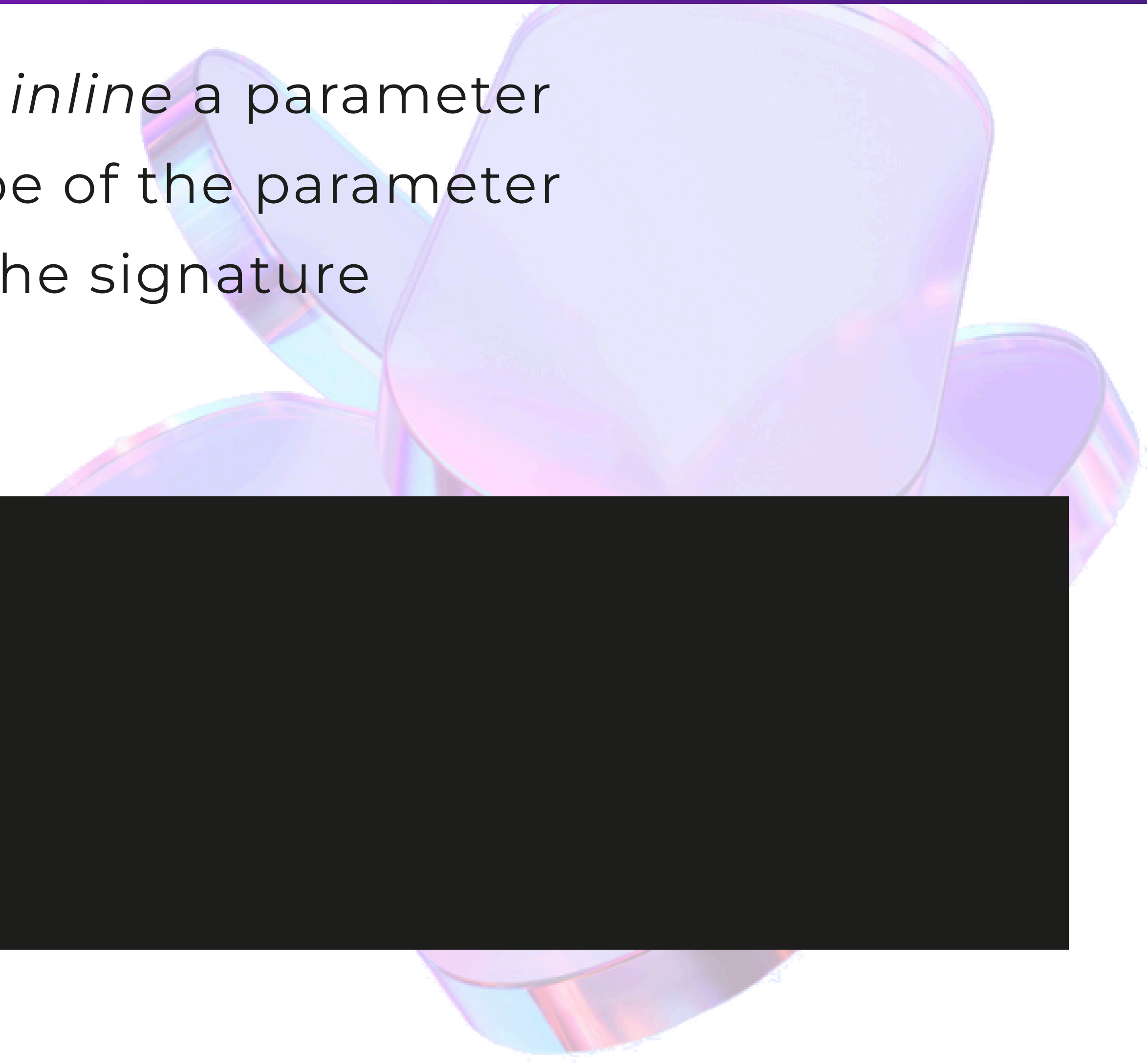
`function` keyword

- ➔ Define an anonymous function
- ➔ Short syntax equivalent to `fun x → match x with`
- ➔ Takes 1 parameter which is implicit

```fsharp
let ouiNon x =
  match x with
  | true  → "Oui"
  | false → "Non"

// Same written with `function`
let ouiNon = function
  | true  → "Oui"
  | false → "Non"
```

☝ Taste matter

Deconstructing parameters



- ➔ As in JavaScript, you can deconstruct *inline* a parameter
- ➔ This is also a way of indicating the type of the parameter
- ➔ The parameter appears unnamed in the signature

Example with a *Record* type 📍

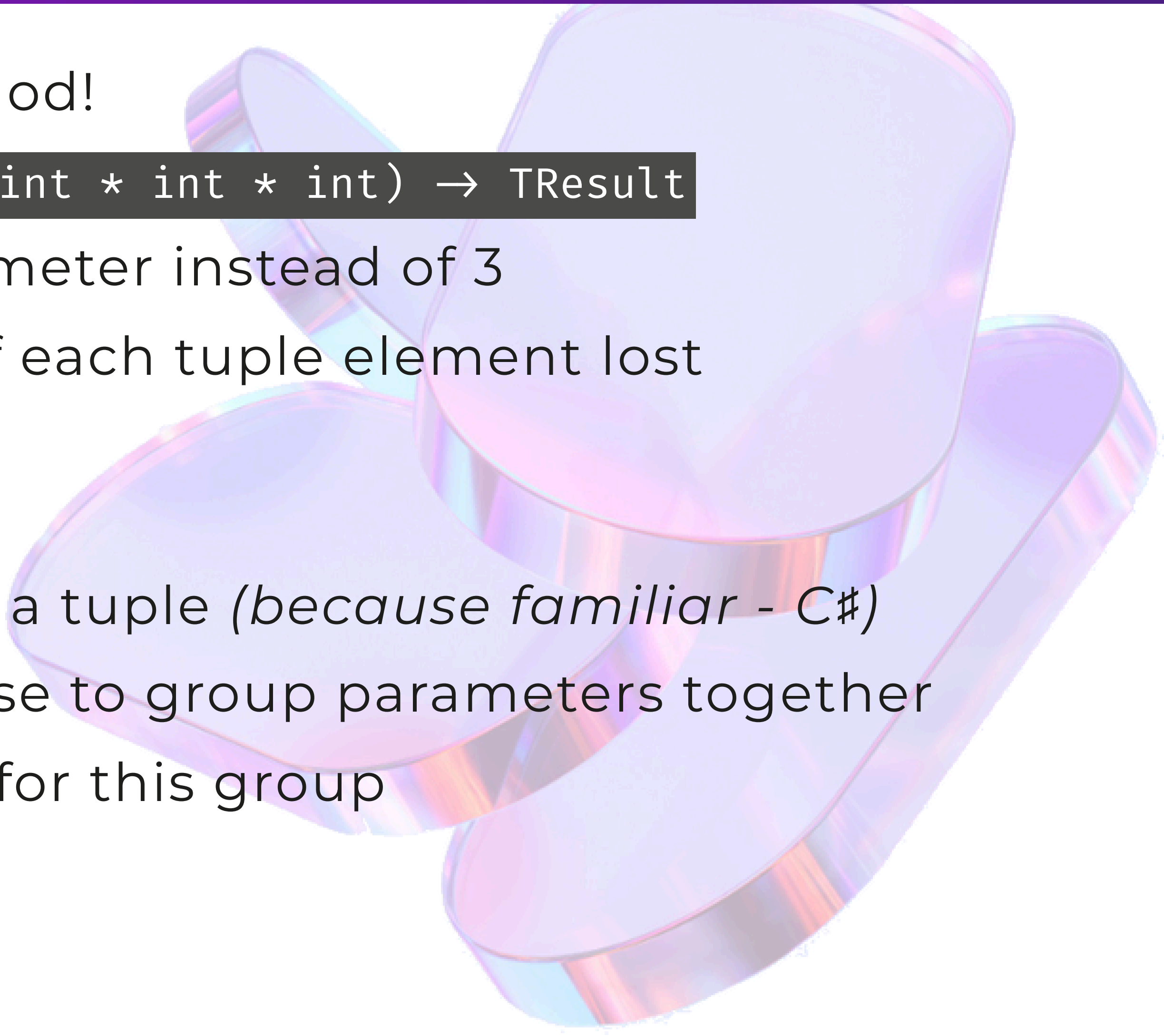
```fsharp
type Person = { Name: string; Age: int }

let name { Name = x } = x      // Person -> string
let age { Age = x } = x        // Person -> int
let age' person = person.Age   // Equivalent explicit

let bob = { Name = "Bob"; Age = 18 } // Person
let bobAge = age bob // int = 18
```

Tuple Parameter

- ➔ As in C♯, you may wish to group function parameters together
  - ➔ For the sake of cohesion, when these parameters form a whole
  - ➔ To avoid *code smell* long parameter list
- ➔ You can group them in a tuple and even deconstruct it

```fsharp
// V1 : too many parameters
let f x y z = ...

// V2 : parameters grouped in a tuple
let f params =
    let (x, y, z) = params
    ...

// V3: same with tuple deconstructed on the spot
let f (x, y, z) = ...
```

Tuple Parameter (2)



- ➔ `f (x, y, z)` looks a lot like a C♯ method!
- ➔ The signature signals the change: `(int * int * int) -> TResult`
  - ➔ The function now has only 1! parameter instead of 3
  - ➔ Possibility of partial application of each tuple element lost

☝️ **Conclusion** :

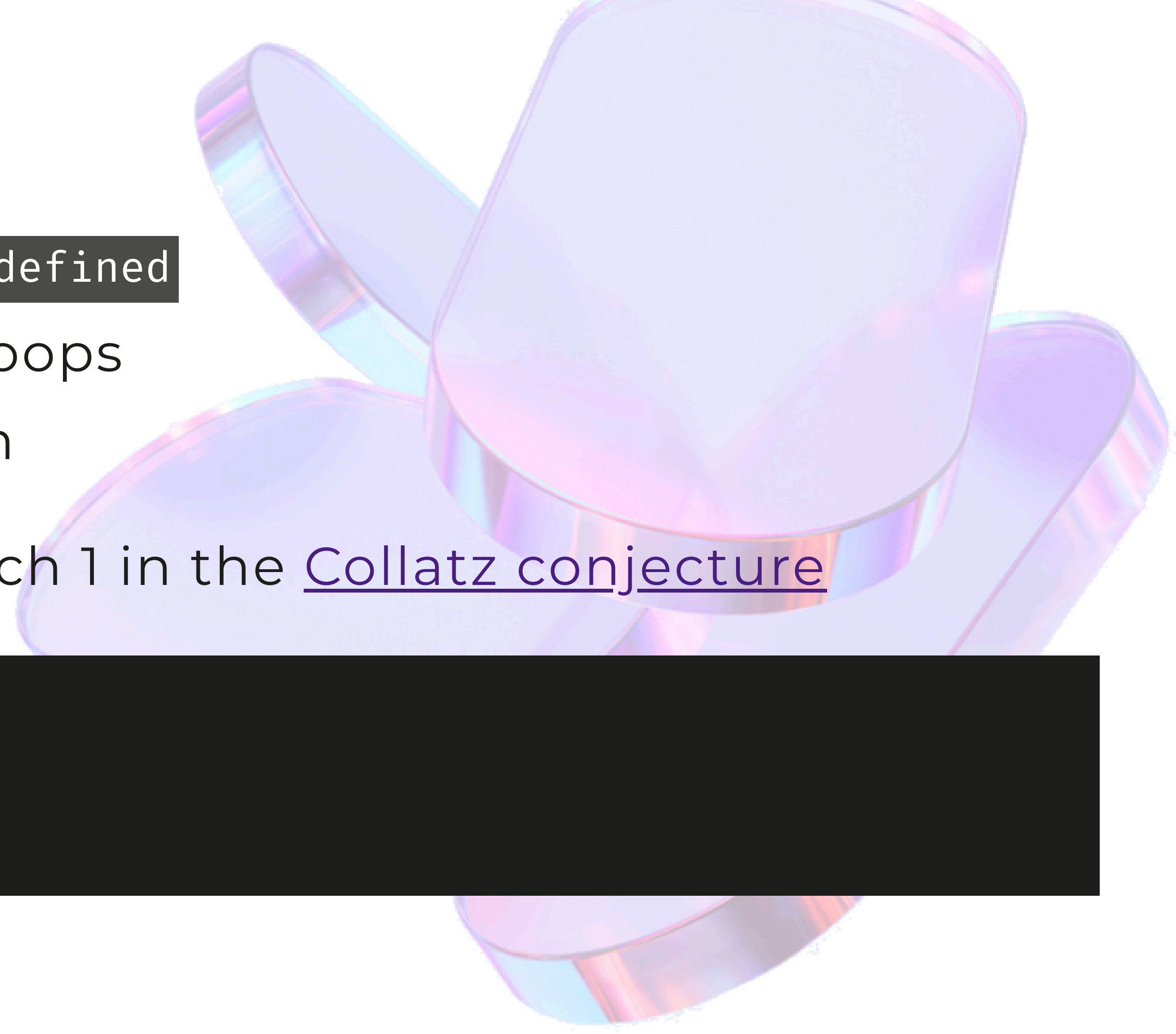
- ➔ Resist the temptation to always use a tuple *(because familiar - C♯)*
- ➔ Reserve this use when it makes sense to group parameters together
  - ➔ Without declaring a specific type for this group

Recursive function

- ➔ Function that calls itself
- ➔ Special syntax with keyword `rec`
  - ➔ otherwise error `FS0039: ... is not defined`
- ➔ Very common in F♯ to replace `for` loops
  - ➔ Because it's often easier to design

Example: find the number of steps to reach 1 in the Collatz conjecture

```fsharp
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)
```



Tail recursion

- ➔ Type of recursion where the recursive call is the last instruction
- ➔ Detected by the compiler and optimized as a loop
  - ➔ Prevents stack overflow
- ➔ Classic method of making tail recursive:
  - ➔ Add an "accumulator" parameter, such as `fold`/`reduce`.

```fsharp
let steps (number: int) : int =
    [<TailCall>] // (F# 8)
    let rec loop count n = // 👉 `loop` = idiomatic name for this type of recursive internal function
        if n = 1         then count
        elif n % 2 = 0 then loop (count + 1) (n / 2)       // 👉 Last instruction = call to `loop`
        else                loop (count + 1) (3 * n + 1)  // 👉 same
    loop 0 number // 👉 Start loop with 0 as initial value for `count`
```

Mutually recursive functions

- ➔ Functions that call each other
- ➔ Must be declared together:
  - ➔ 1st function indicated as recursive with `rec`
  - ➔ other functions added to declaration with `and`

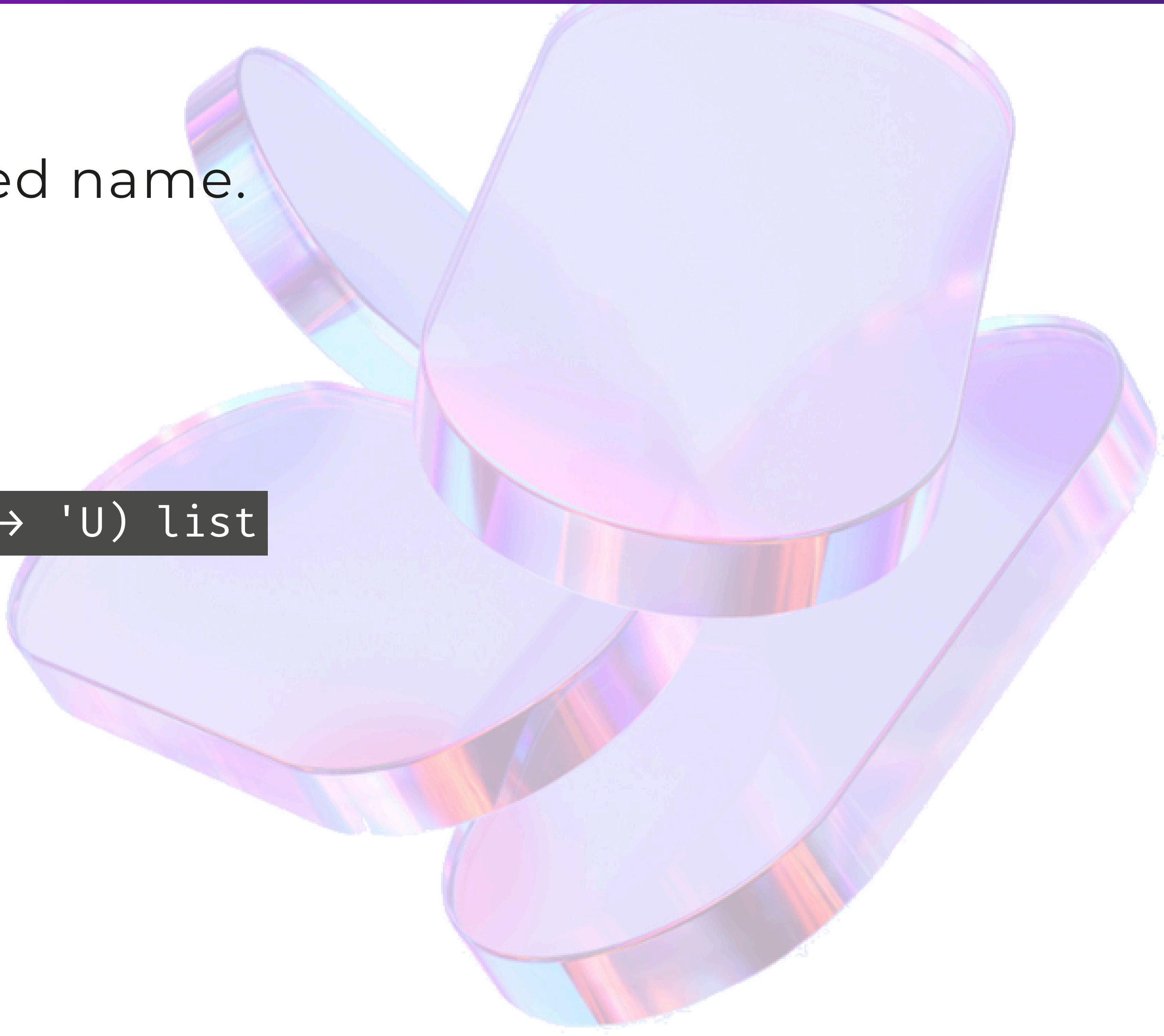
```
// ⚠️ Convoluted algo, just for illustration purposes

let rec Even x =          // 👉 Keyword `rec`
    if x = 0 then true
    else Odd (x-1)        // 👉 Call to `Odd` defined below
and Odd x =               // 👉 Keyword `and`
    if x = 0 then false
    else Even (x-1)       // 👉 Call to `Even` defined above
```

Function overload

- ➔ A function cannot be overloaded!
- ➔ Each version should have a dedicated name.

Example:

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



Template function

Create specialized "overloads" · Example: wrap `String.Compare`:

```fsharp
type ComparisonResult = Bigger | Smaller | Equal // Union type 📍

let private compareTwoStrings (comparison: StringComparison) string1 string2 =
    let result = System.String.Compare(string1, string2, comparison)
    if result > 0 then
        Bigger
    elif result < 0 then
        Smaller
    else
        Equal

// Partial application of the 'comparison' parameter
let compareCaseSensitive   = compareTwoStrings StringComparison.CurrentCulture
let compareCaseInsensitive = compareTwoStrings StringComparison.CurrentCultureIgnoreCase
```

Template function (2)

☝️ The additional parameter is placed at a different location in C♯ and F♯:

➔ Comes last in C♯:

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

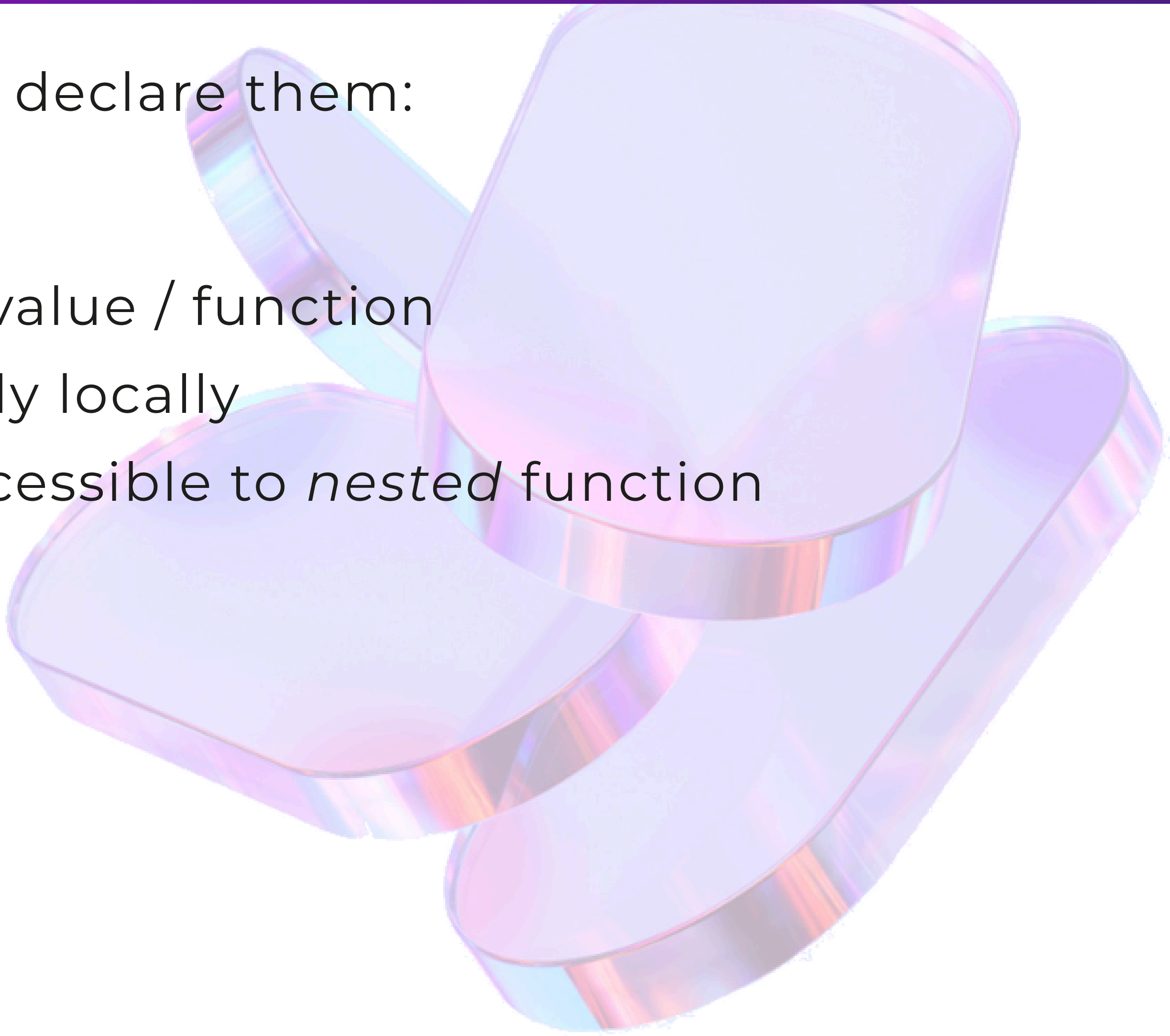
➔ Comes first in F♯, to enable its partial application:

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

Function Organisation

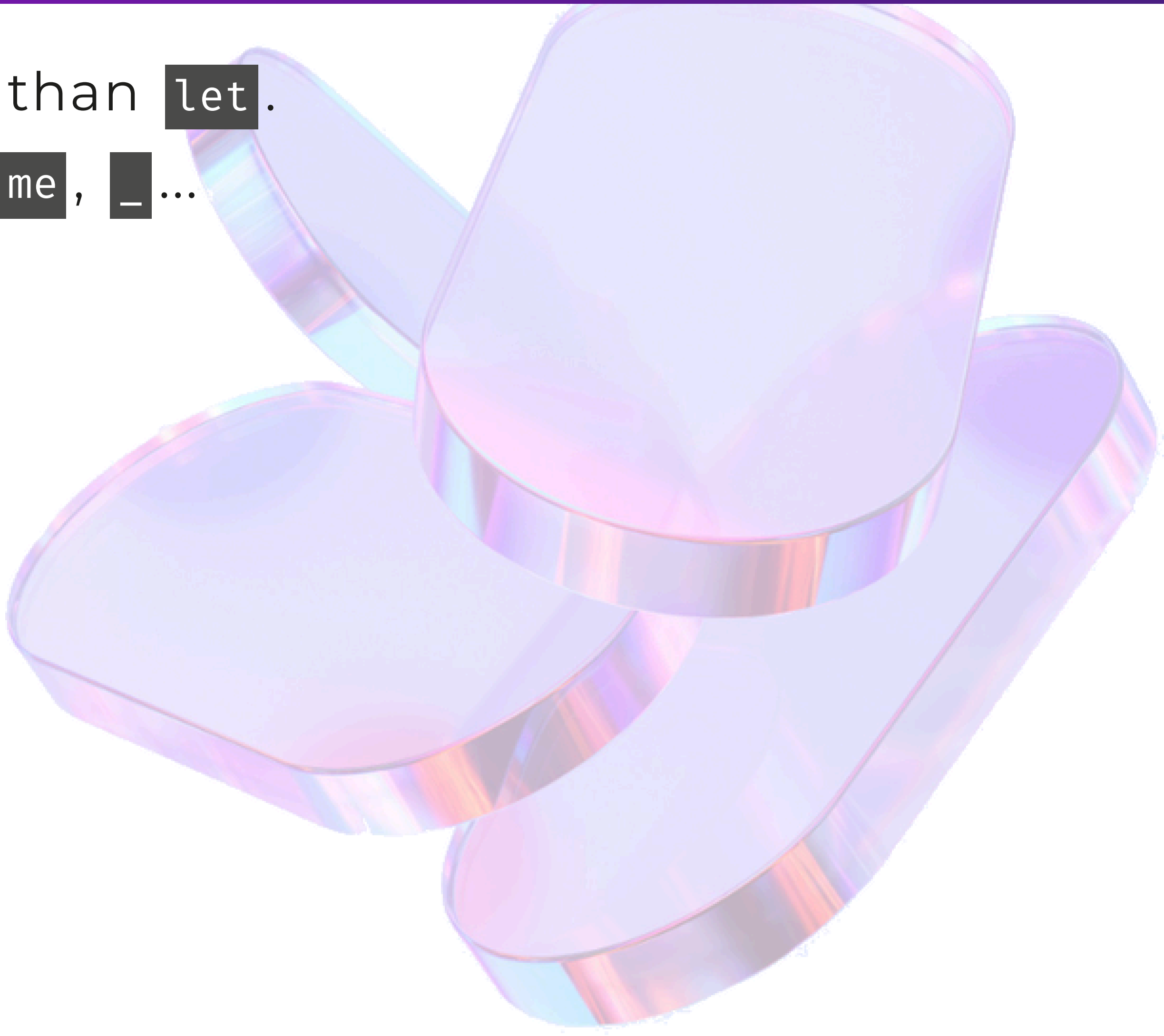
3 ways to organize functions = 3 places to declare them:

- ➔ *Module* 📍
- ➔ *Nested* : function declared inside a value / function
  - ➔ 💡 Encapsulating helpers used only locally
  - ➔ ☝️ Parent function parameters accessible to *nested* function
- ➔ *Method* : type member *(next slide)*



Methods

- ➔ Defined with keyword `member` rather than `let`.
- ➔ Choice of *self-identifier*: `this`, `self`, `me`, `_`…
- ➔ Choice of parameters:
  - ➔ Tuplified: OOP style
  - ➔ Curried: FP style



Methods - Example

```fsharp
type Product =
    { SKU: string; Price: float }

    // Tuple style
    member this.TupleTotal(qty, discount) =
        (this.Price * float qty) - discount

    // Curried style
    member me.CurriedTotal qty discount = // 👉 `me` / "this"
        (me.Price * float qty) - discount // 👉 `me.Price` to access the `Price` property
```

Function *vs* Method

Feature	Function	Method
Naming convention	camelCase	PascalCase
Currying	✅ yes	✅ if not tuplified nor overridden
Named parameters	❌ no	✅ if tuplified
Optional parameters	❌ no	✅ if tuplified
Overload	❌ no	✅ if tuplified

Function *vs* Method (2)

Feature	Function	Method
Parameter inference (declaration)	➖ Possible	➖ yes for `this`, possible for the other parameters
Argument inference (usage)	✅ yes	❌ no, object type annotation needed
High-order function argument	✅ yes	➖ yes with shorthand member, no with lambda otherwise
`inline` supported	✅ yes	✅ yes
Recursive	✅ yes with `rec`	✅ yes

Standard functions

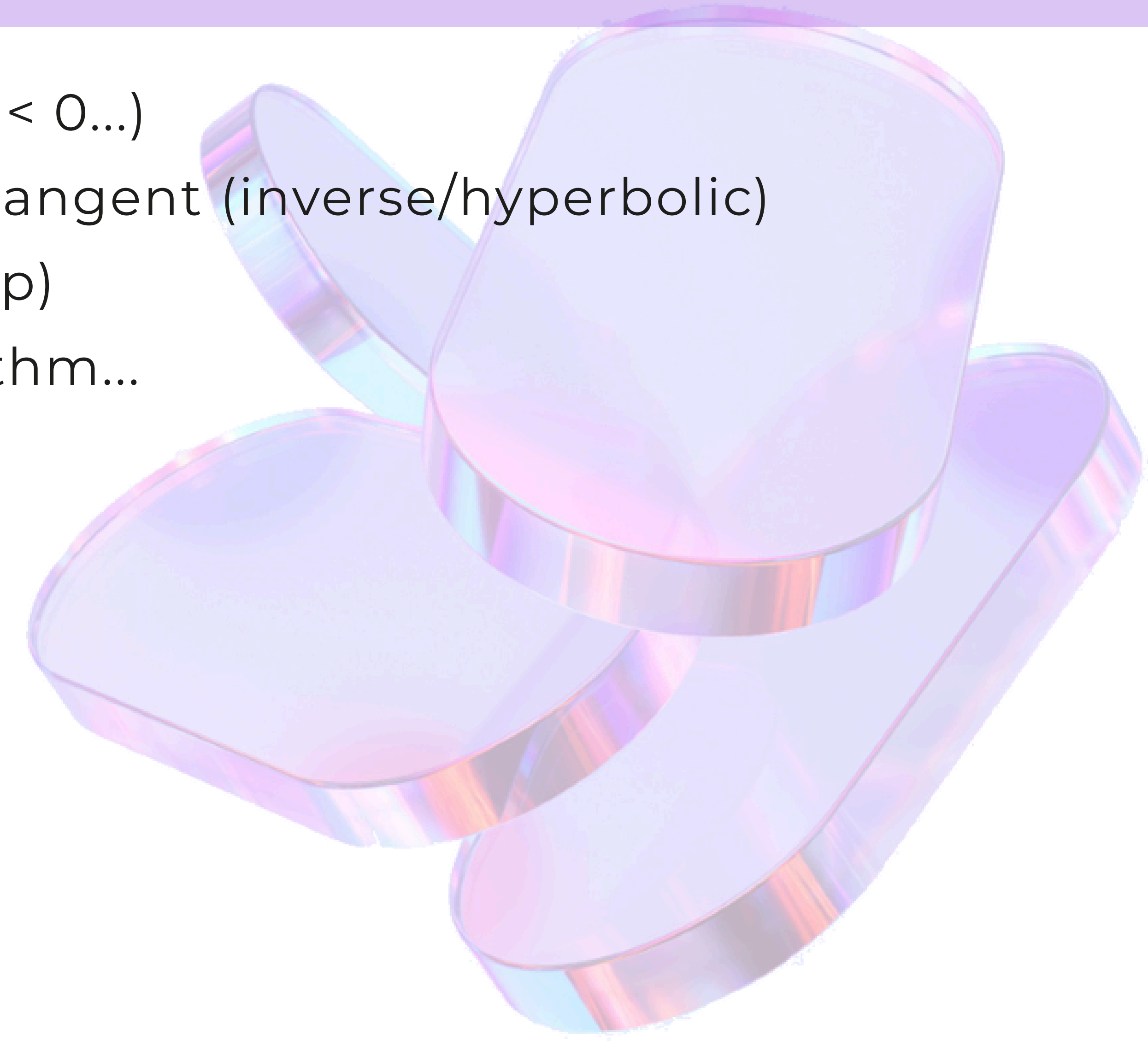
Defined in `FSharp.Core` automatically imported

Conversion

- ➔ `box`, `tryUnbox`, `unbox` : *boxing*, *unboxing* (attempt)
- ➔ `byte`, `char`, `decimal`, `float`, `int`, `string` : conversion to `byte`, `char`, ...
- ➔ `enum<'TEnum>` : conversion to given enum type

Math

- ➔ `abs`, `sign`: absolute value, sign (-1 if < 0...)
- ➔ `(a)cos(h)`, `(a)sin`, `(a)tan`: (co)sinus/tangent (inverse/hyperbolic)
- ➔ `ceil`, `floor`, `round`: rounding (inf, sup)
- ➔ `exp`, `log`, `log10`: exponential, logarithm...
- ➔ `pown x (n: int)` : `x` to the power `n`.
- ➔ `sqrt` : square root



Misc

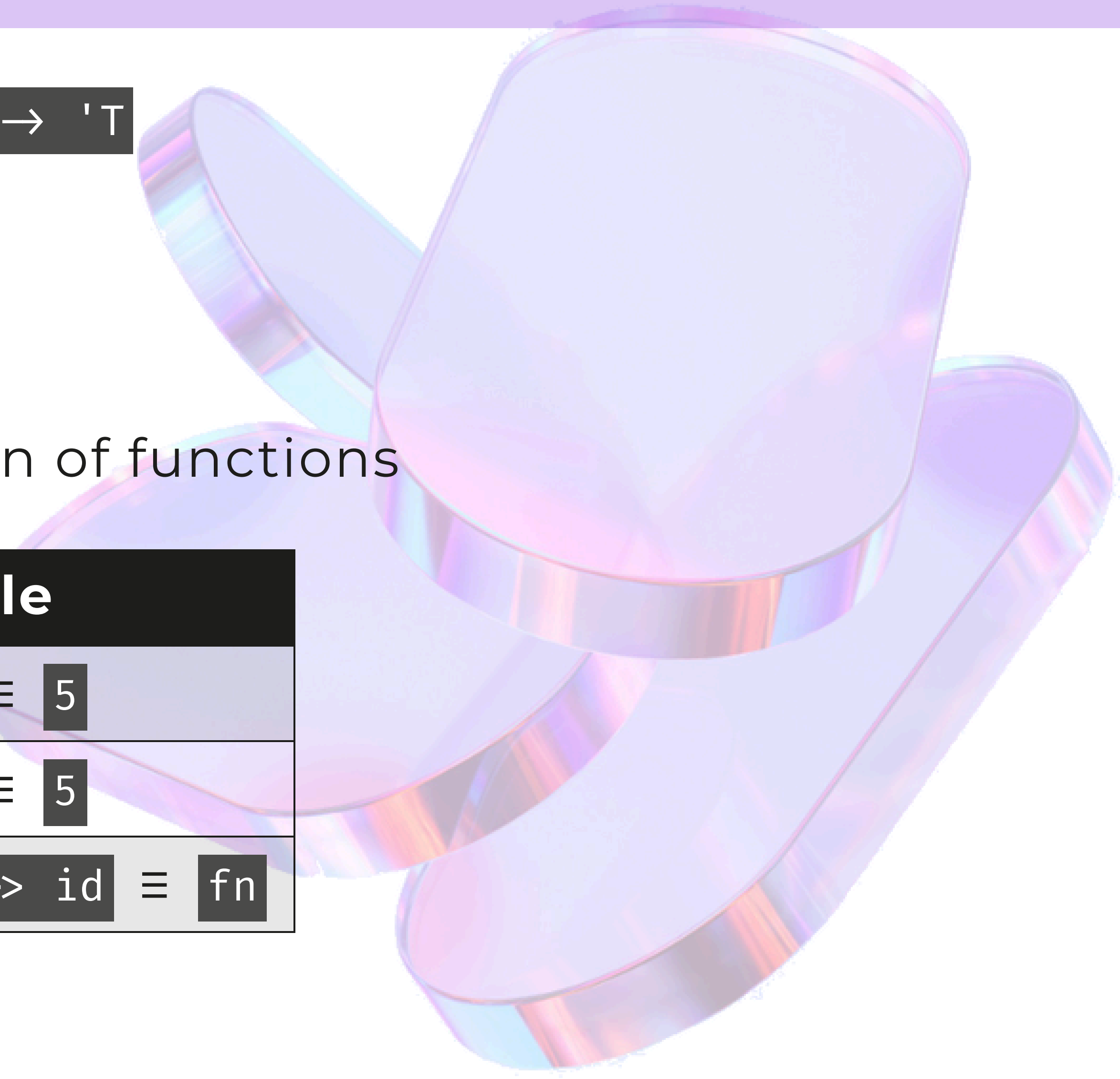
- ➔ `compare a b : int`: returns -1 if a < b, 0 if =, 1 if >
- ➔ `hash`: calculates hash (code)
- ➔ `max`, `min`: maximum and minimum of 2 comparable values
- ➔ `ignore`: to swallow/skip a value, return `()` (`unit`)
- ➔ `id` : next slide 👇

`id` : identity

Definition `let id x = x` · Signature : `(x: 'T) → 'T`
→ Single input parameter function
→ Only returns this parameter

Why such a function ❓
→ Zero / Neutral element in the composition of functions

Operation	Identity	Example
Addition `+`	`0`	`0 + 5` ≡ `5 + 0` ≡ `5`
Multiplication `*`	`1`	`1 * 5` ≡ `5 * 1` ≡ `5`
Composition `>>`	`id`	`id >> fn` ≡ `fn >> id` ≡ `fn`



`id` - Use cases

With a *high-order function* doing 2 things:

➔ 1 operation

➔ 1 value mapping via param `'T → 'U`.

Ex: `List.collect fn list` = flatMap: flatten + map

How to do just the operation and no mapping?

➔ `list ▷ List.collect (fun x → x)` 👎

➔ `list ▷ List.collect id` 👍

➔ ☝️ Best alternative: `List.concat list` 💯

3. The operators

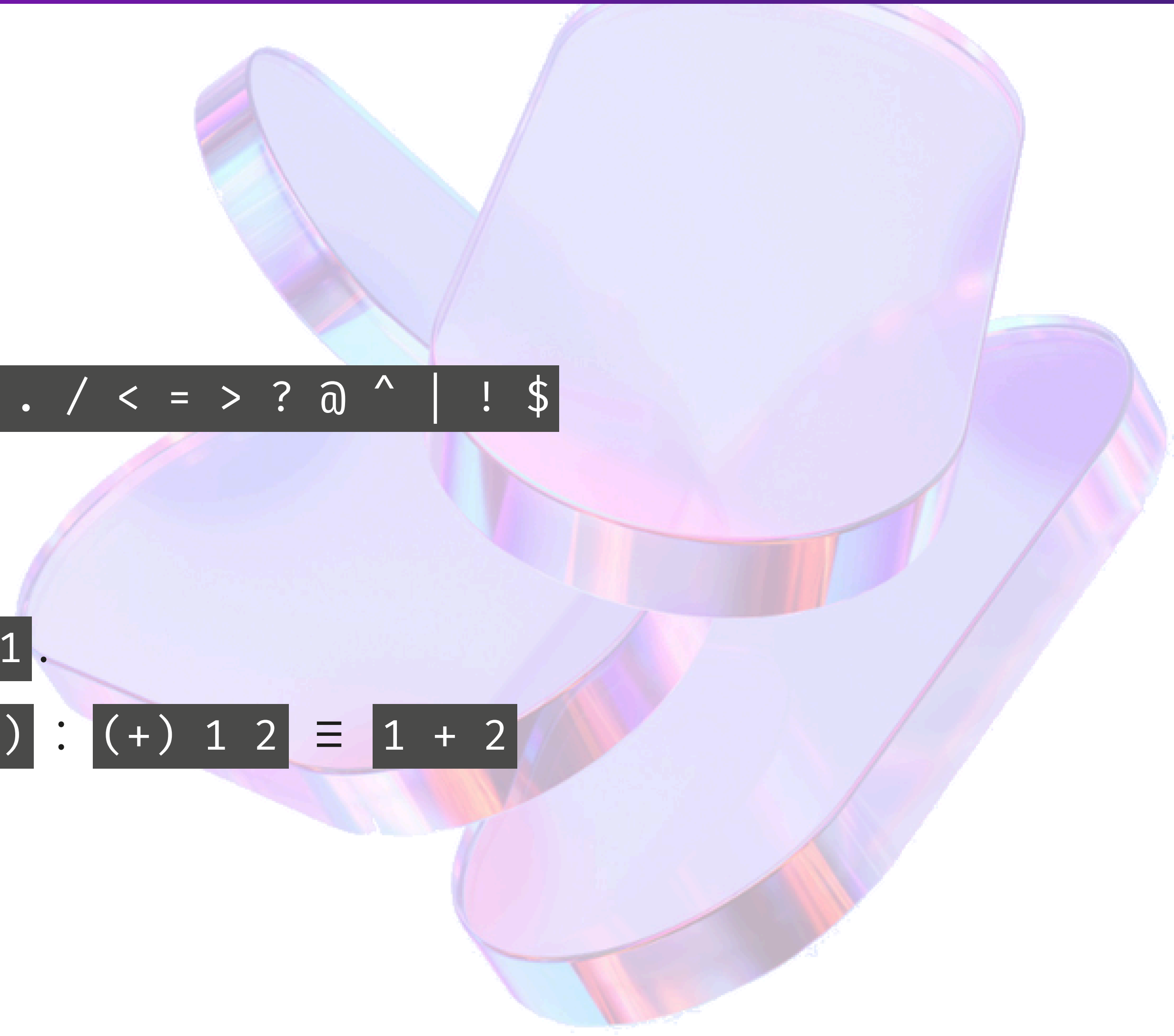
Operator

Is defined as a function

- ➔ Unary operator: `let (~symbols) = ...`
- ➔ Binary operator: `let (symbols) = ...`
- ➔ *Symbols* = combination of `% & * + - . / < = > ? @ ^ | ! $`

2 ways to use operators

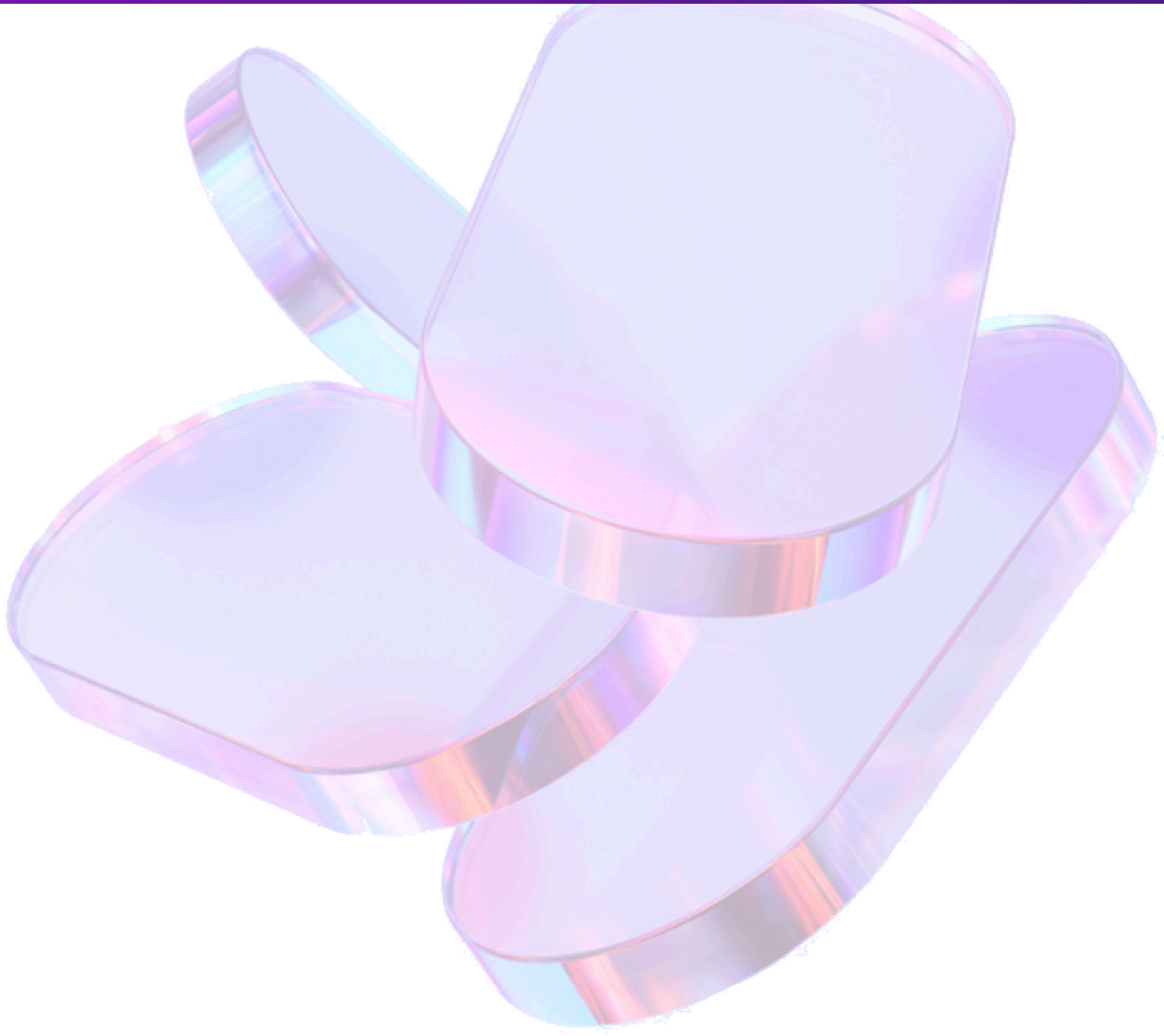
- ➔ As operator → infix `1 + 2` or prefix `-1`.
- ➔ As a function → symbols between `()` : `(+) 1 2` ≡ `1 + 2`



Standard operators

Defined in `FSharp.Core`

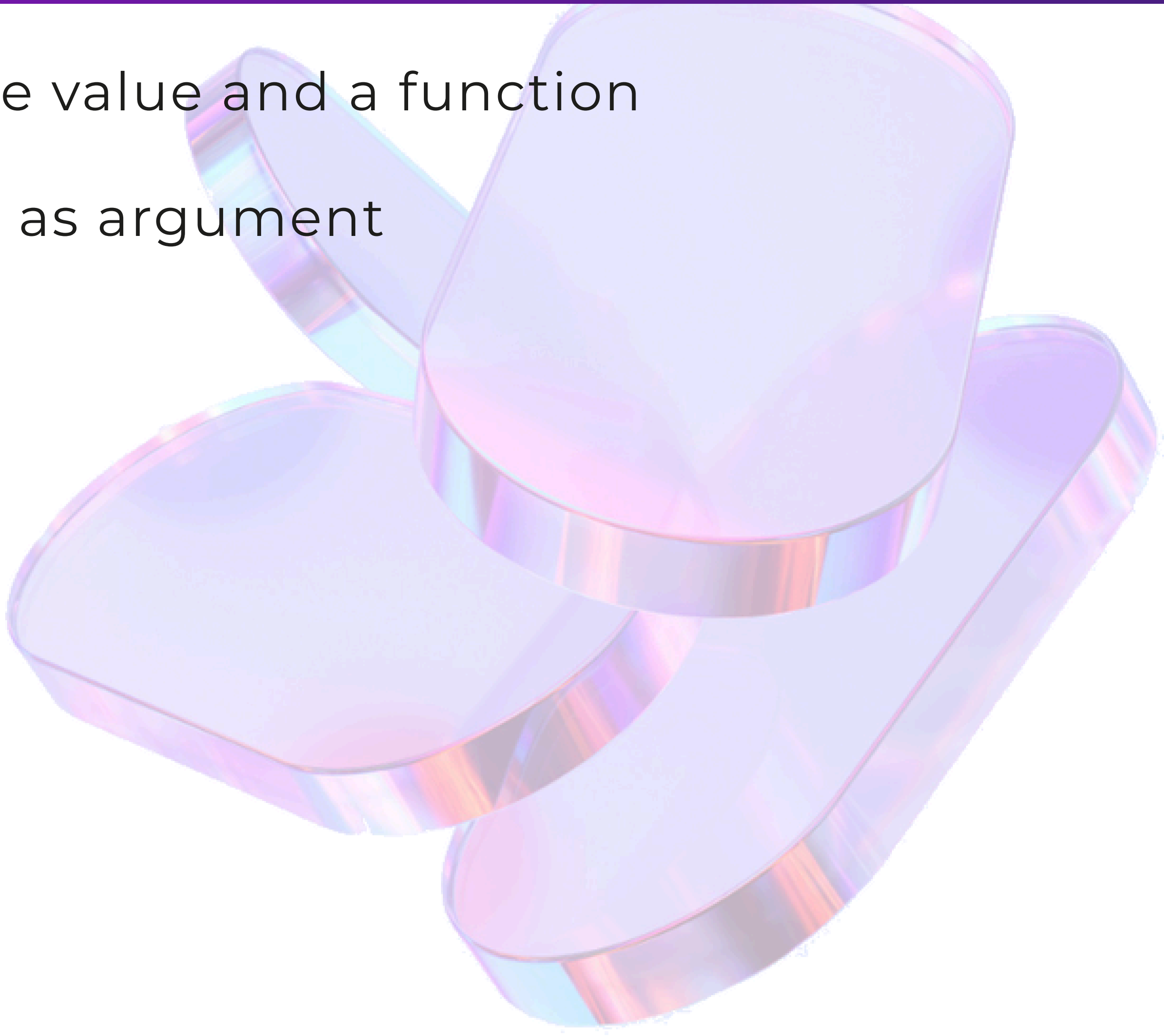
- ➔ Arithmetic operators: `+`, `-`…
- ➔ Pipeline operators
- ➔ Composition operators



Pipe operators

Binary operators, placed between a simple value and a function

- ➔ Apply value to function = Pass value as argument
- ➔ Avoid parentheses / precedence
- ➔ There are several *pipes*
  - ➔ *Pipe right* `▷` : the "classic" *pipe*.
  - ➔ *Pipe left* `◁` a.k.a. *inverted* pipe
  - ➔ *Pipe right 2* `|▷`
  - ➔ Etc.



Operator *Pipe right* `▷`

Reverses the order between function and value: `val ▷ fn` ≡ `fn val`

- ➔ Natural "subject-verb" order, as a method call of an object (`obj.M(x)`)
- ➔ *Pipeline*: chain function calls, without intermediate variable
- ➔ Object inference help. Example:

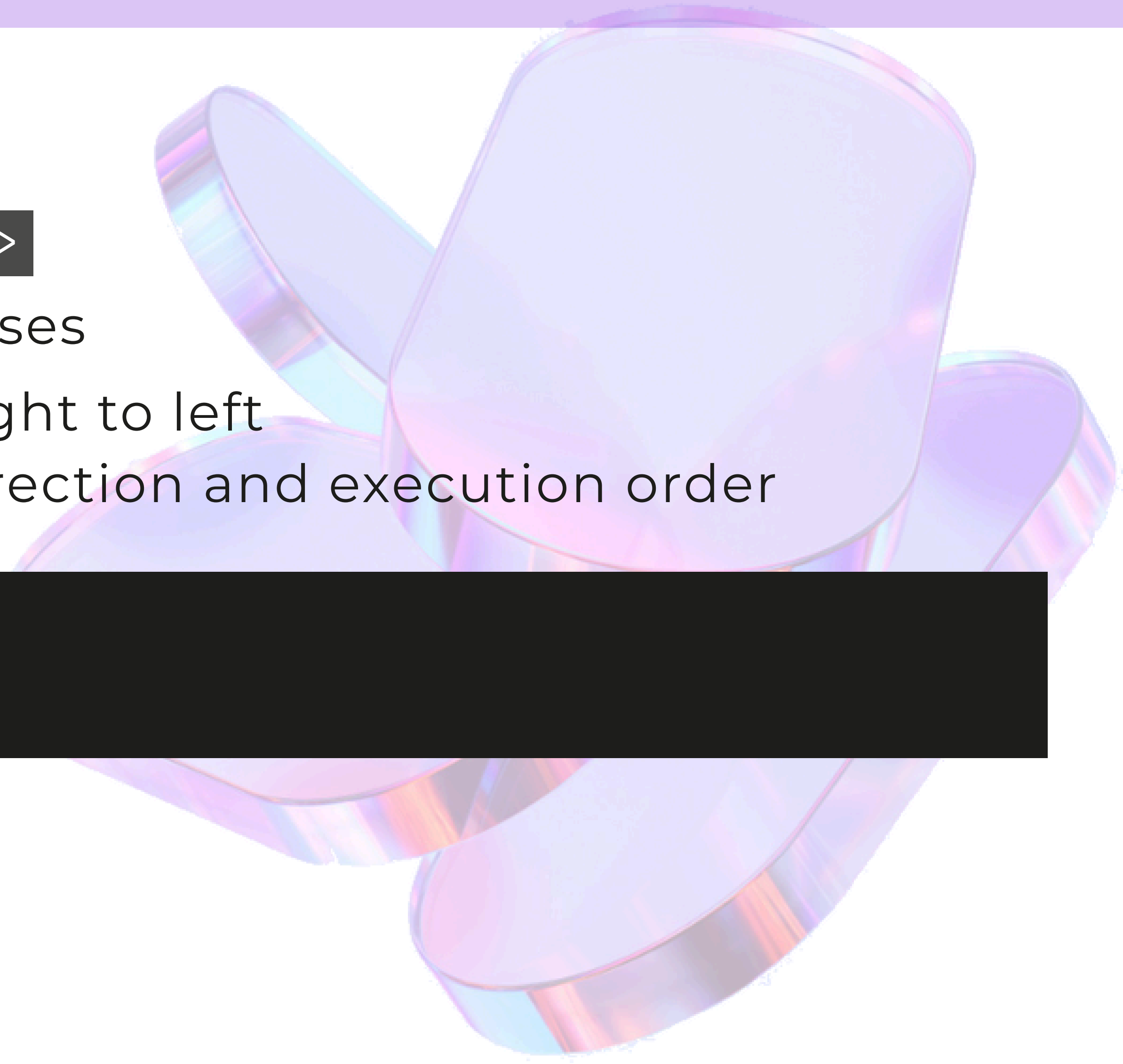
```
let items = ["a"; "bb"; "ccc"]

let longestKo = List.maxBy (fun x → x.Length) items  // ❌ Error FS0072
//                                  ~~~~~~~~

let longest = items ▷ List.maxBy (fun x → x.Length) // ✅ Works, returns "ccc"
```

Operator *Pipe left* `◁`

`fn ◁ expression` ≡ `fn (expression)`

- ➔ ☝️ Usage a little less common than `▷`
- ➔ ✅ Minor advantage: avoids parentheses
- ➔ ❌ Major disadvantage: reads from right to left
  → Reverses natural English reading direction and execution order

```fsharp
printf "%i" 1+2 // 💥 Error
printf "%i" (1+2) // With brackets
printf "%i" ◁ 1+2 // With inverted pipe
```



Operator *Pipe left* `◁` (2)

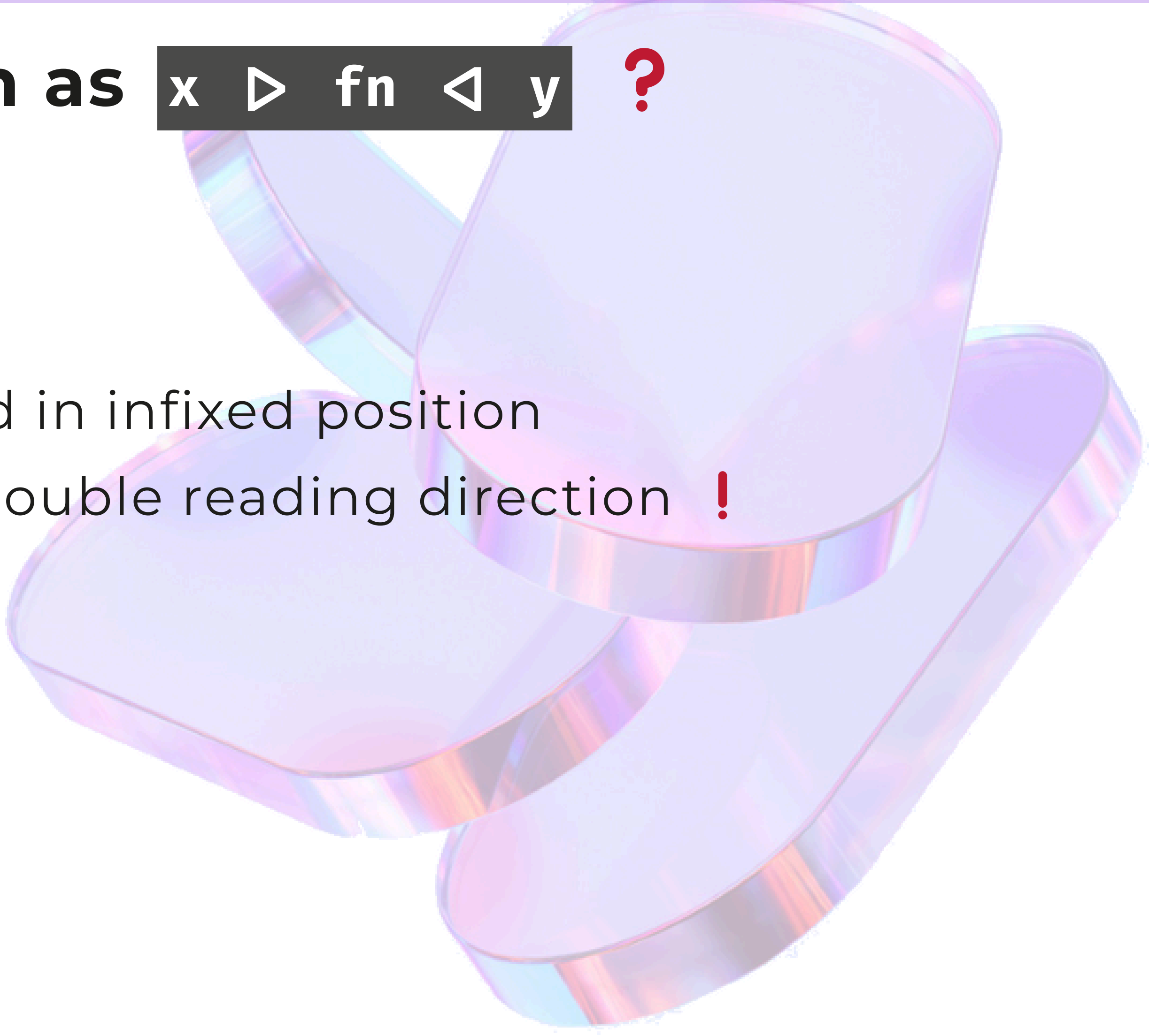
What about an expression such as `x ▷ fn ◁ y` ?

Executed from left to right:

`(x ▷ fn) ◁ y` ≡ `(fn x) ◁ y` ≡ `fn x y`

- ➔ In theory: would allow `fn` to be used in infixed position
- ➔ In practice: difficult to read due to double reading direction ❗

👉 Tip: **TO BE AVOIDED**



Operator *Pipe right 2* `||>`

`(x, y) ||> fn` ≡ `fn x y`

- To pass 2 arguments at once, as a tuple
- Used infrequently, for example with `fold` to pass list & seed

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

// 😕 Difficult to spot the seed, at the far right
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)

// 💡 Replace lambda with named function
let addIfEven acc x = if x % 2 = 0 then acc + x else acc
let sumOfEvens'' = items |> List.fold addIfEven 0
```

Compose operator `>>`

Binary operators placed **between two functions**
→ The result of the 1st function will serve as an argument to the 2nd function

`f >> g` ≡ `fun x → g (f x)` ≡ `fun x → x ▷ f ▷ g` a `fun x → x ▷ f ▷ g`

⚠️ Types must match: `f: 'T → 'U` and `g: 'U → 'V`
→ We get a signature function `'T → 'V`

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

let add1Times2 x = times2(add1 x) // 😕 Explicit style but + busy
let add1Times2' = add1 >> times2 // 👍 Concise style
```

Operator Compose inverse `<<`

Rarely used, except to restore a natural order of terms

Example with operator `not` (which replaces the `!` in C♯):

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

// Classic pipeline
let Odd x = x ▷ Even ▷ not

// Rewritten with inverse composition
let Odd = not << Even
```

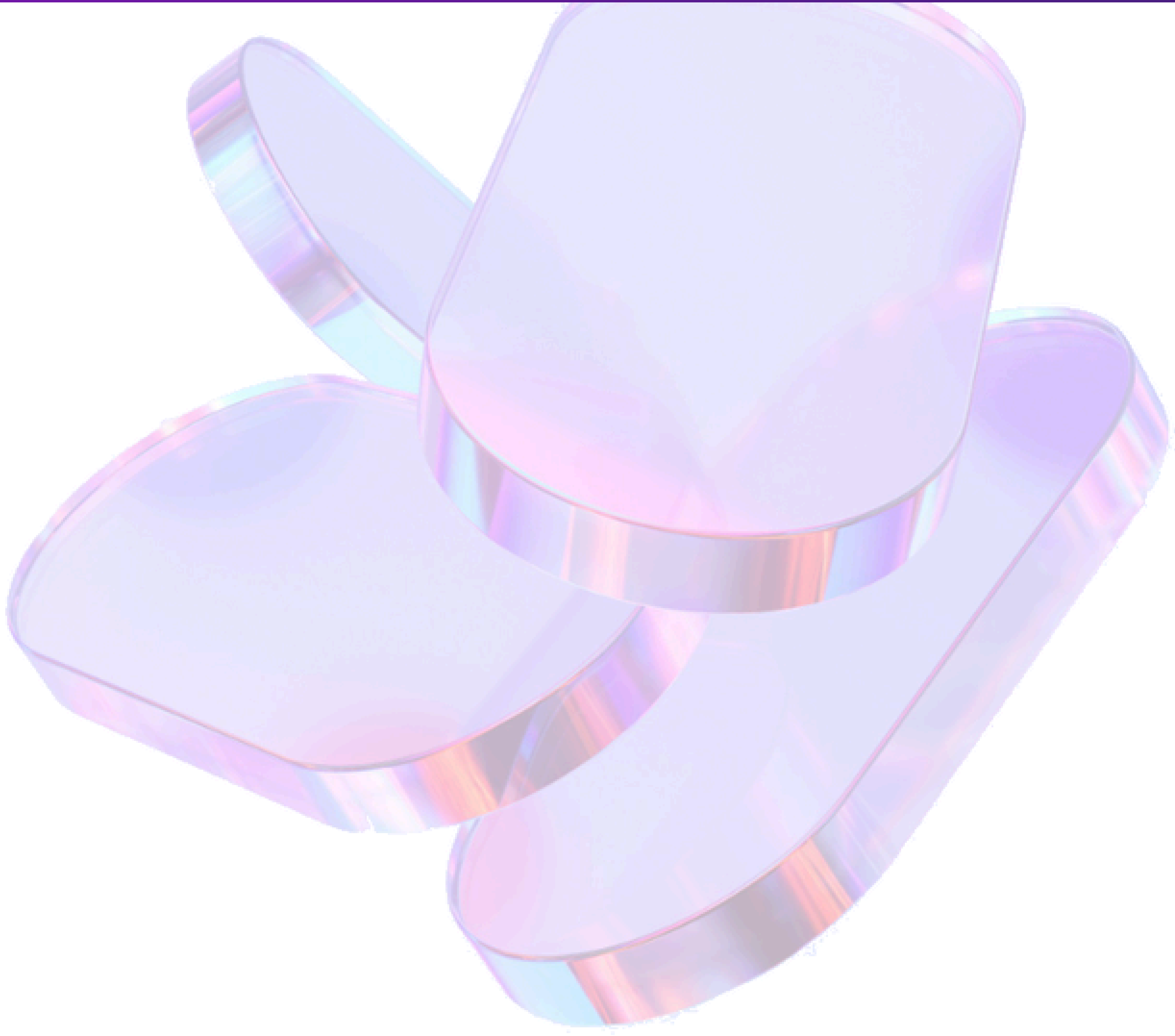
Pipe `▷` or Compose `>>` ?

Compose `let h = f >> g`

- ➔ Reasoning at function level

Pipe `let result = value ▷ f`

- ➔ Reasoning at value level

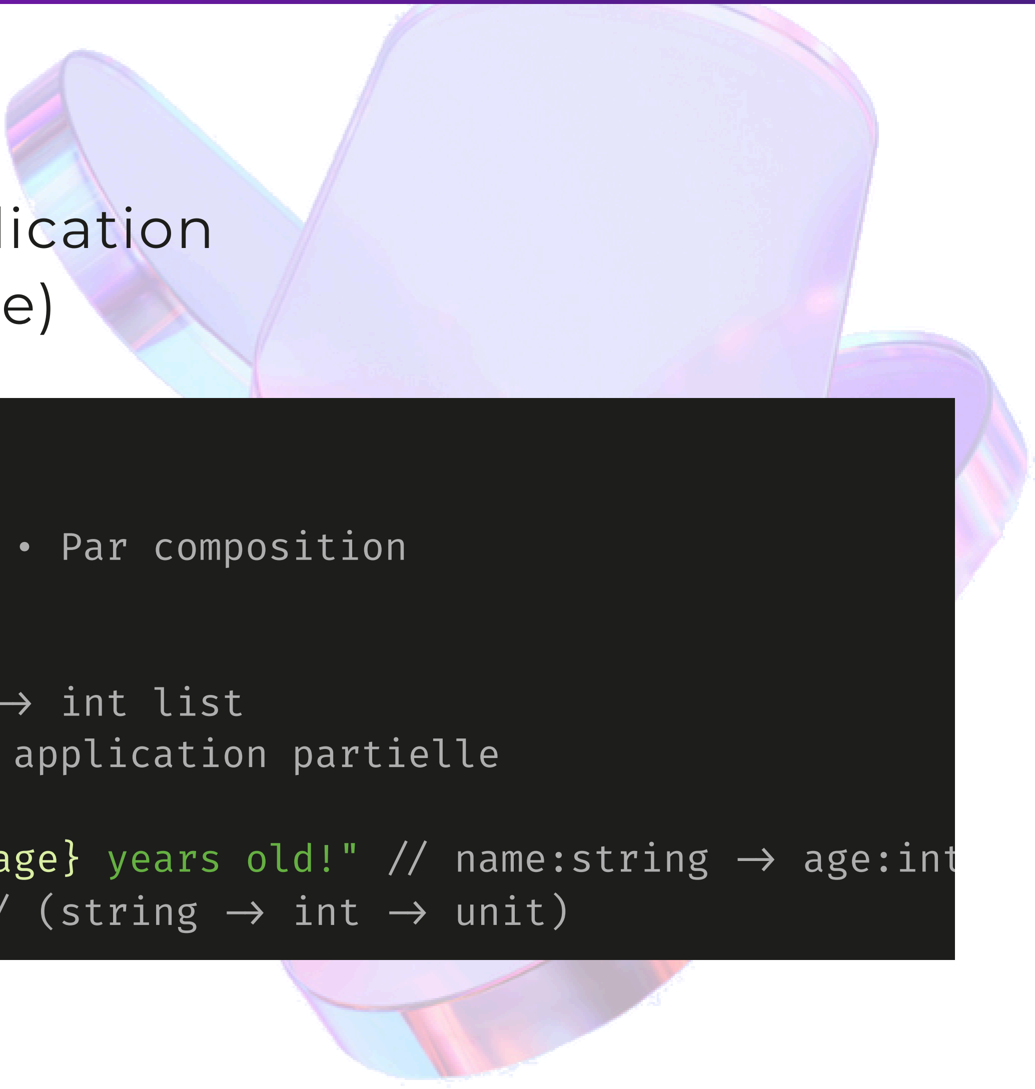


Point-free style

A.k.a *Tacit Programming*

Function defined by composition or partial application
→ **Implicit parameter**, hence `point-free` (in space)

```fsharp
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
let greet' = printfn "My name is %s and I am %d years old!" // (string → int → unit)
```



Point-free - Pros/Cons ⚖️

✅ Pros

Concise style - Abstract parameters, operate at function level

❌ Cons

Loses the name of the parameter now implicit in the signature
👌 fine if the function remains understandable
(due to parameter types + the function name)
👌 fine in a narrow scope
otherwise can obfuscate the code - imagine in a code review!
❌ not recommended for a public API

Point-free - Limit 🛑

Works poorly with generic functions :

```
let isNotEmptyKo = not << List.isEmpty                // 💥 Error FS0030: Value restriction
let isNotEmpty<'a> = not << List.isEmpty<'a>          // 👌 With type annotation
let isNotEmpty' list = not (List.isEmpty list)        // 👌 Style explicit
```

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

Fonction `inline` : principle

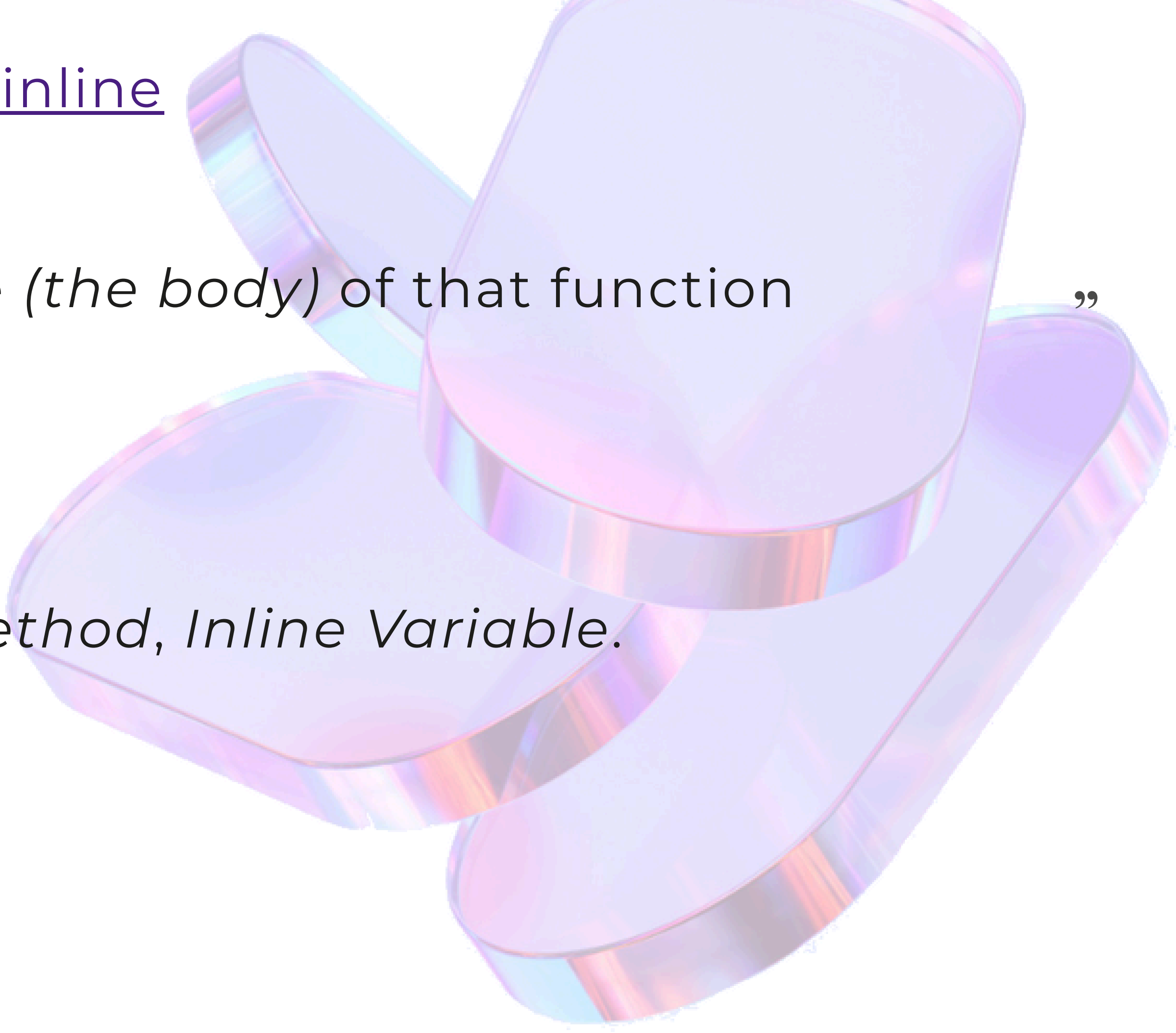
🔗 https://fr.wikipedia.org/wiki/Extension_inline

> **compilation inlining:**
> → replaces a function call with the code *(the body)* of that function

➔ Performance gain 👍

➔ Longer compilation time ⚠️

💡 Same principle as refactoring *Inline Method*, *Inline Variable*.



`inline` (2)

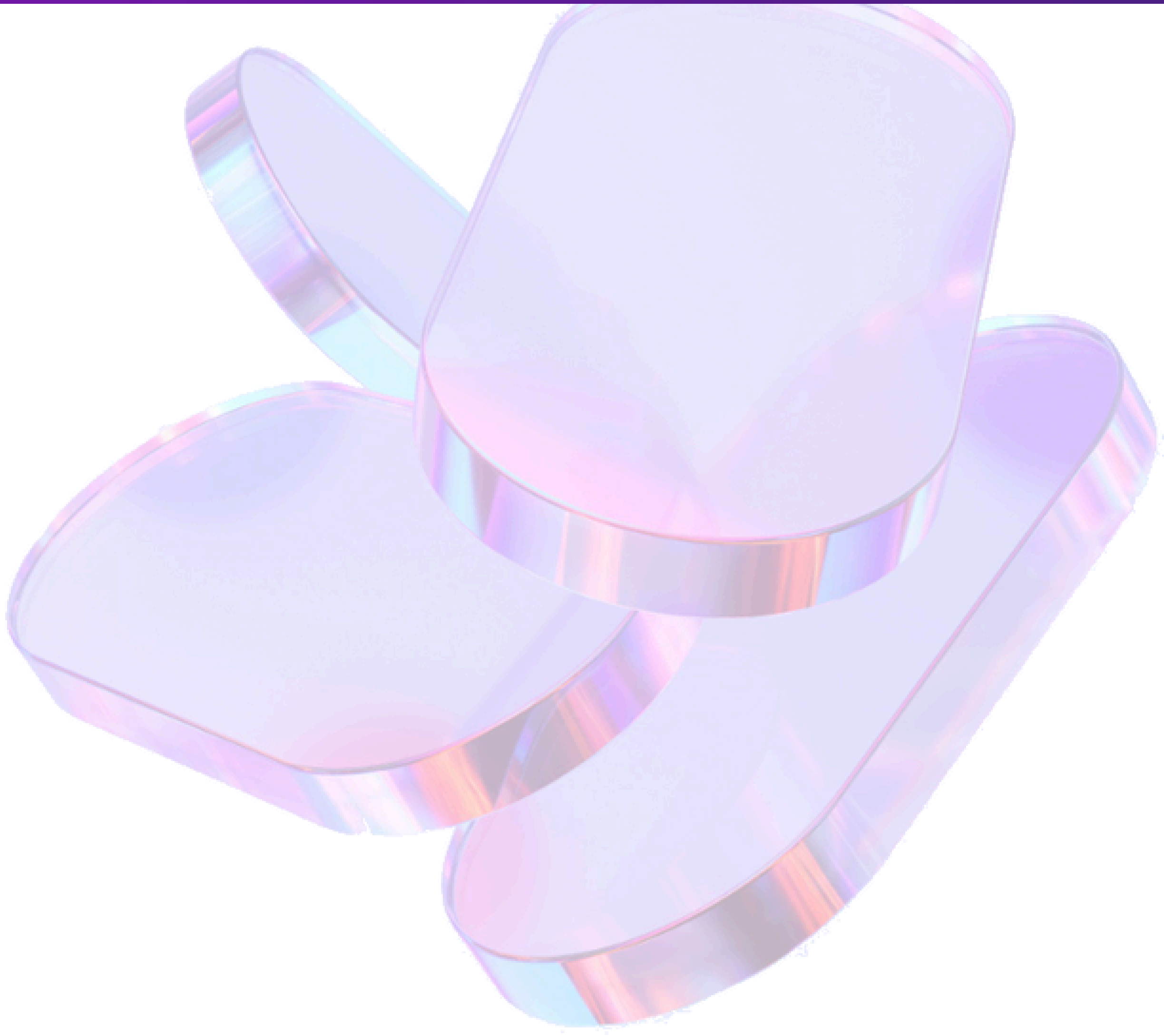
Keyword `inline` tells the compiler to *"inline "* the function
→ Typical use: small "syntactic sugar" function/operator

```
// 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 // inline pipe
     ~= ()          // inline ignore
```

Custom operators

2 possibilities :

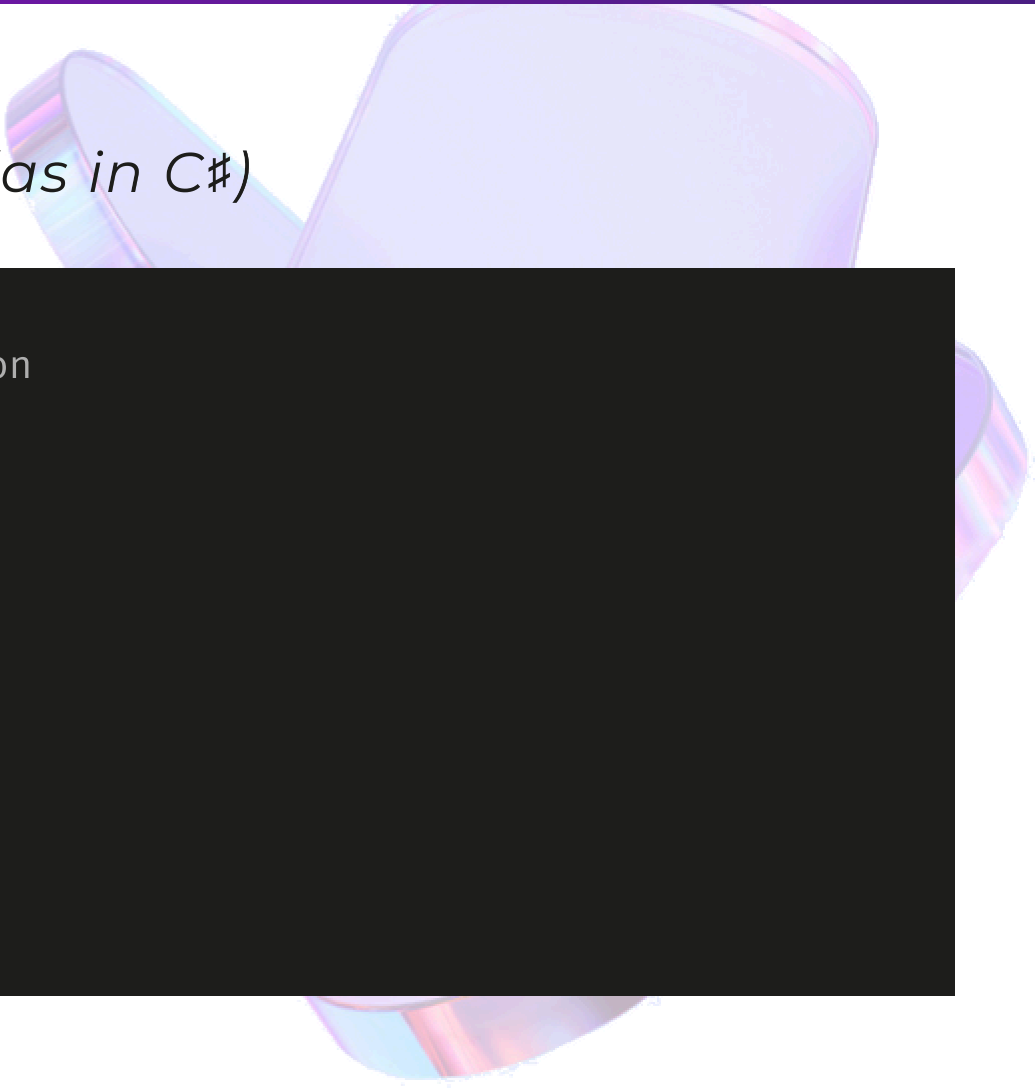
- ➔ Operator overload
- ➔ Creation of a new operator



Operator overload

Generally concerns a specific type
→ Overload defined within the associated type *(as in C♯)*

```fsharp
type Vector = { X: int; Y: int } with
    // Unary operator (cf ~ and 1! param) for vector inversion
    static member (~-) (v: Vector) =
        { X = -v.X
          Y = -v.Y }

    // Binary addition operator for 2 vectors
    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 }
```

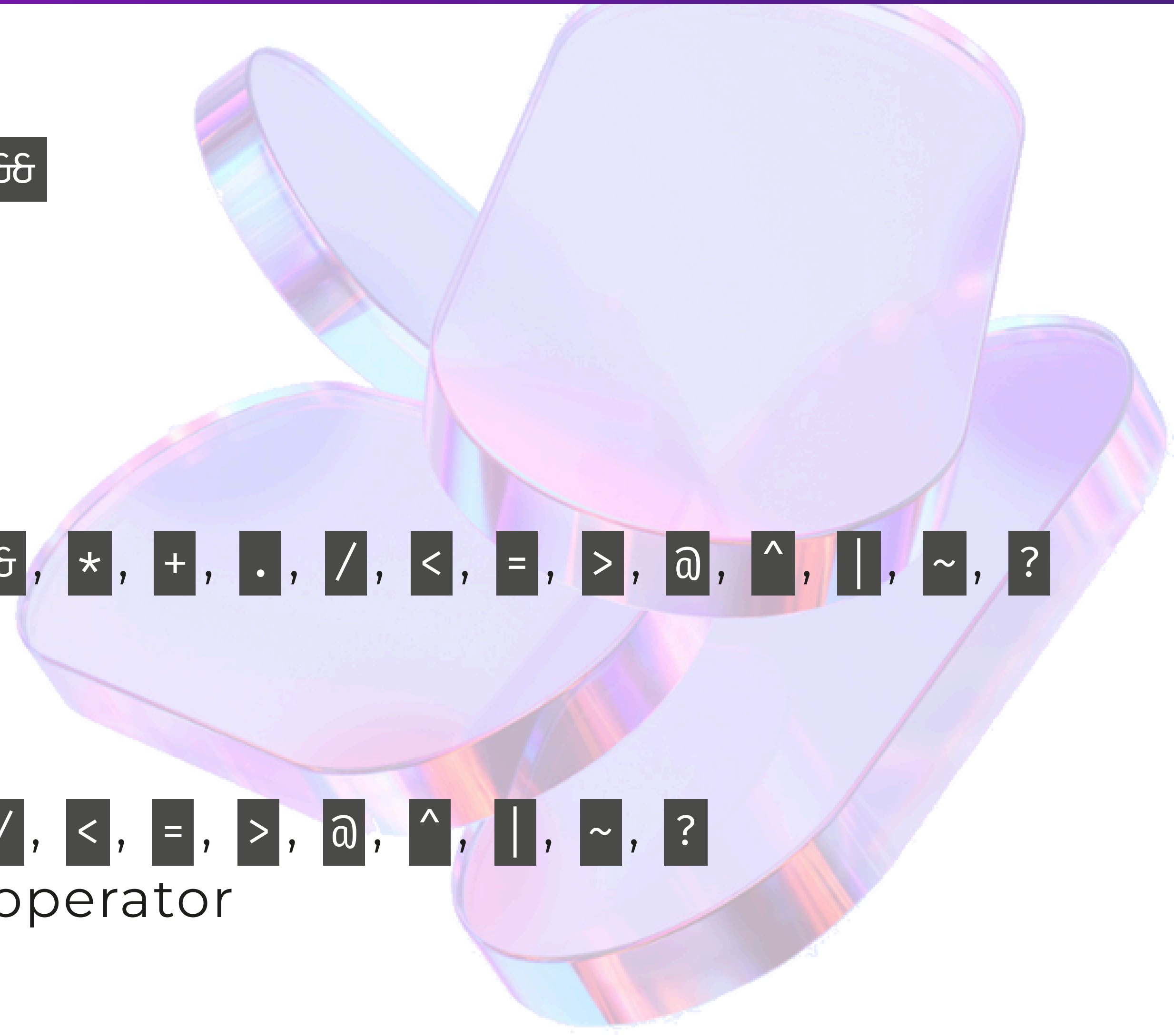


Creation of a new operator

- ➔ Definition rather in a module or in an associated type
- ➔ Classic use case: alias for existing function, used as infix

```fsharp
// "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"
```

Symbols allowed in an operator



Unary "tilde " operator
→ `~` followed by `+`, `-`, `+.`, `-.`, `%`, `%%`, `&`, `&&`

Unary operator "snake "
→ Several `~`, e.g. `~~~~`

Unary operator "bang "
→ `!` followed by a combination of `!`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `=`, `>`, `@`, `^`, `|`, `~`, `?`
→ Except `≠` (!=) which is binary

Binary operator
→ Any combination of `!`, `%`, `&`, `*`, `+`, `.`, `/`, `<`, `=`, `>`, `@`, `^`, `|`, `~`, `?`
→ which does not correspond to a unary operator

Usage symbols

All operators are used as is

❗ Except the unary operator "tilde": used without the initial `~`.

Operator	Declaration	Usage
Unaire tilde	`let (~&&) x = …`	`&&x`
Unaire snake	`let (~~~) x = …`	`~~~x`
Unaire bang	`let (!!!) x = …`	`!!!x`
Binary	`let (<^>) x y = …`	`x <^> y`

Operator or function?

Infix operator *vs* function

👍 **Pros** :

➔ Respects the natural reading order (left → right)

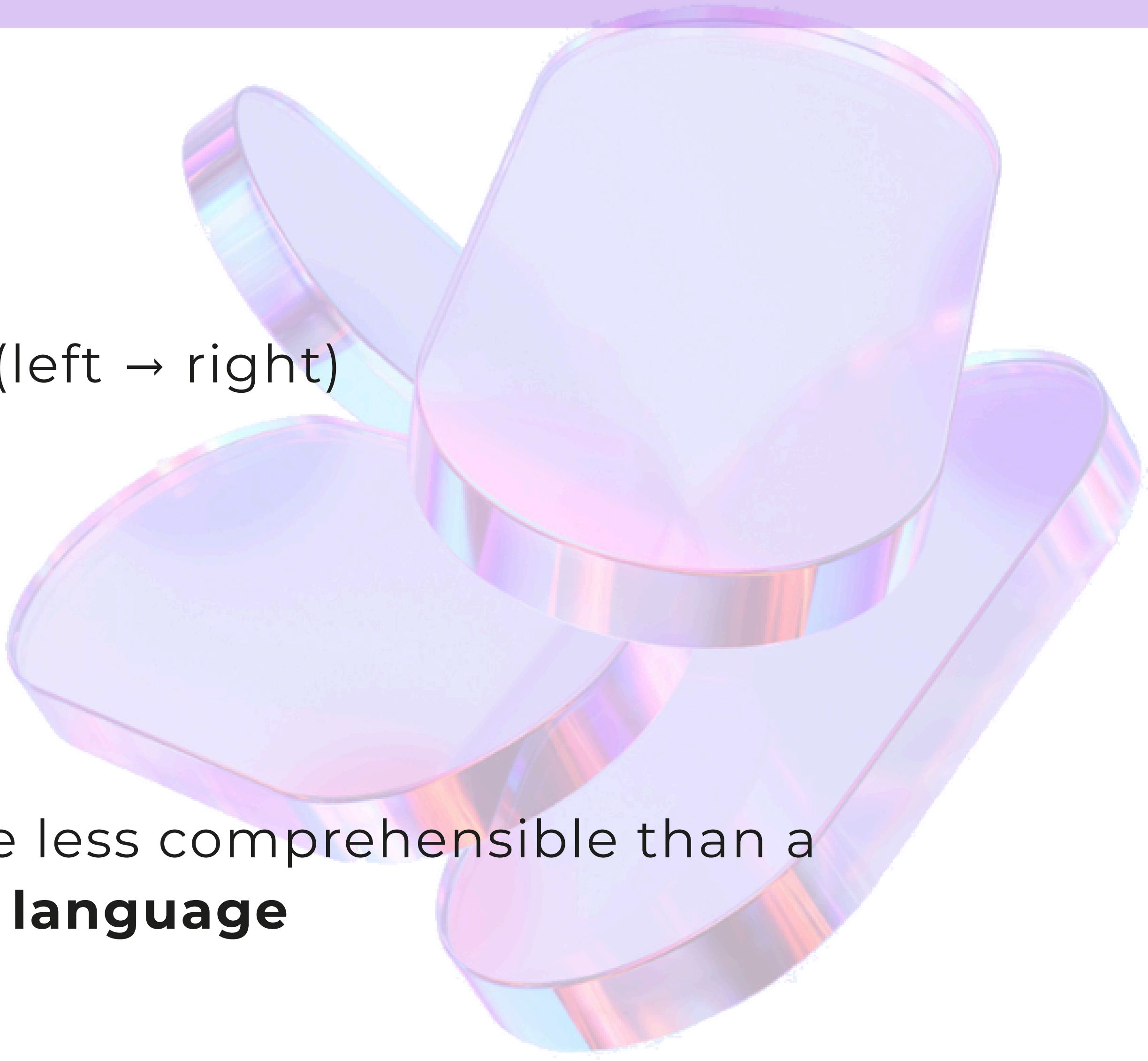
➔ avoids parentheses

→ `1 + 2 * 3` *vs* `multiply (add 1 2) 3`

→ `1 + 2 * 3` *vs* `multiply (add 1 2) 3`

⚠️ **Cons** :

➔ A "folkloric" operator (e.g. `@!`) will be less comprehensible than a function whose name uses the **domain language**



Using an operator as a function

💡 You can use the partial application of a binary operator :

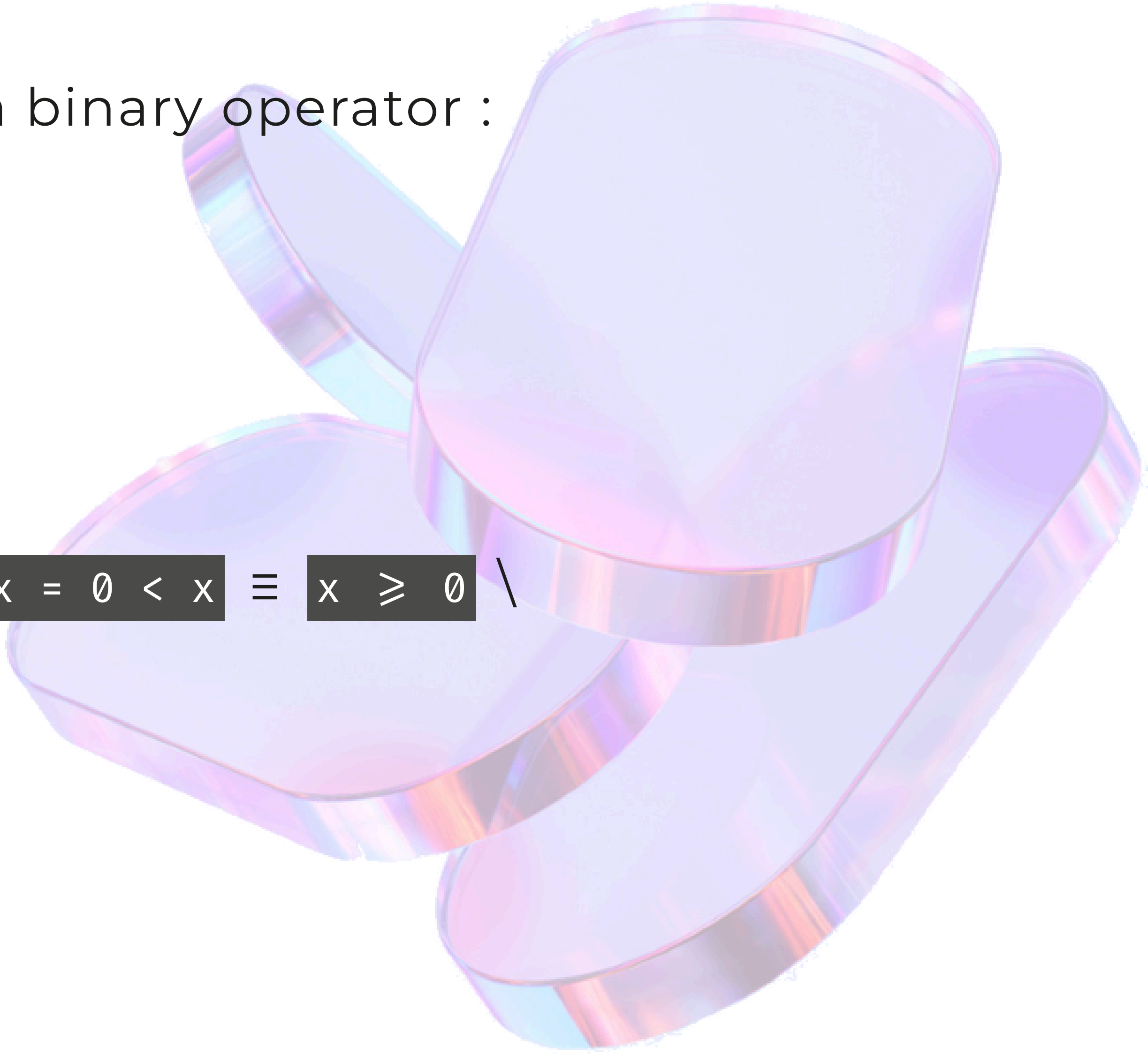
Examples:

➔ Instead of a lambda:

→ `(+) 1` ≡ `fun x → x + 1`

➔ To define a new function :

→ `let isPositive = (<) 0` ≡ `let isPositive x = 0 < x` ≡ `x ≥ 0` \



4. Interop with the .NET BCL

BCL = Base Class Library .NET

void method

A .NET `void` method is seen in F# as returning `unit`.

```
let list = System.Collections.Generic.List<int>()
list.Add
(* IntelliSense Ionide:
abstract member Add:
   item: int
      -> unit
*)
```

Conversely, an F# function returning `unit` is compiled into a `void` method.

Calling a BCL method with N arguments

A .NET method with several arguments is "pseudo-tuplified":

- ➔ All arguments must be specified (1)
- ➔ Partial application of parameters is not supported (2)
- ➔ Calls don't work with a real F♯ tuple ⚠️ (3)

```
System.String.Compare("a", "b") // ✅ (1)
System.String.Compare "a","b"   // ❌
System.String.Compare "a"       // ❌ (2)

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

# `out` Parameter - In C♯

`out` used to have multiple output values from a method
→ Ex : `Int32.TryParse`, `Dictionary<,>.TryGetValue` :

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

`out` Parameter - In F♯

Output can be consumed as a tuple 👍

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

Instantiate a class with `new`?

- ➔ Class constructors are regular functions in F♯ 🤩
- ➔ `new` keyword is supported but not recommended

```fsharp
type MyClass(i) = class end

let c1 = MyClass(12)      // 👍
let c2 = new MyClass(234) // 👌 OK but not idiomatic

let cs = [1..3] ▷ List.map MyClass // High-order functions
```

`new` keyword for `IDisposable`

- ➔ `new` keyword is required to instantiate `IDisposable`
- ➔ Compiler warning otherwise

```fsharp
open System.IO

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

Calling an overloaded method

- ➔ Compiler may not understand which overload is being called
- ➔ Tips: call with named argument

```fsharp
let createReader fileName =
    new System.IO.StreamReader(path = fileName)
    // ☝️ Param `path` → `filename` inferred as `string`

let createReaderByStream stream =
    new System.IO.StreamReader(stream = stream)
    // ☝️ Param `stream` of type `System.IO.Stream`
```

5. 🍔 Quiz

Question 1

1. How to define the return value (`v`) of a function (`f`)?

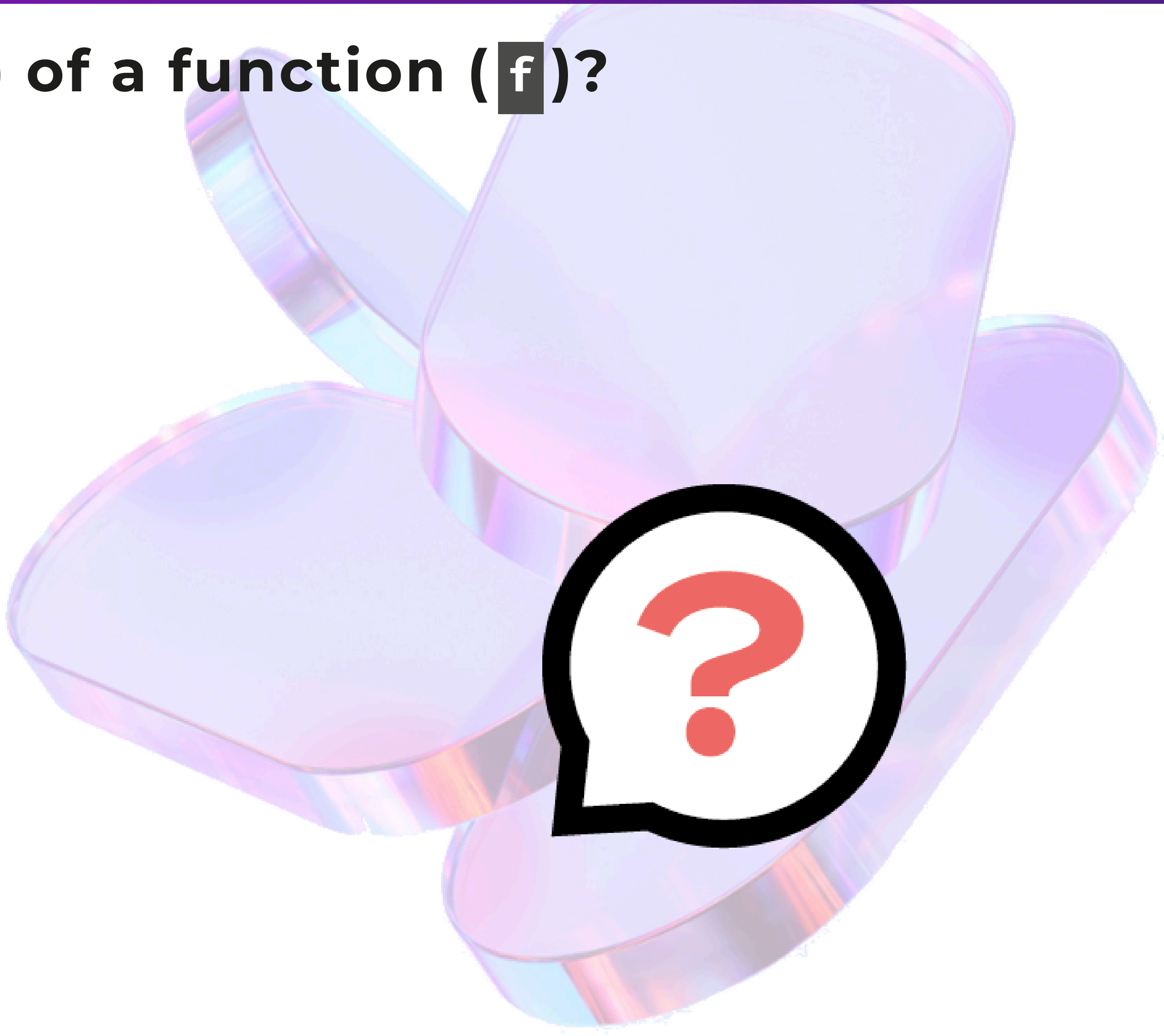
A. Simply name the value `result`.

B. End the function with `return v`.

C. Do `f = v`

D. `v` is the last line of `f`.

⏱️ 10’’



Answer 1

1. How to define the return value (`v`) of a function (`f`)?

A. Simply name the value `result`. ❌

B. End the function with `return v`. ❌

C. Do `f = v`. ❌

D. `v` is the last line of `f`. ✅

Question 2

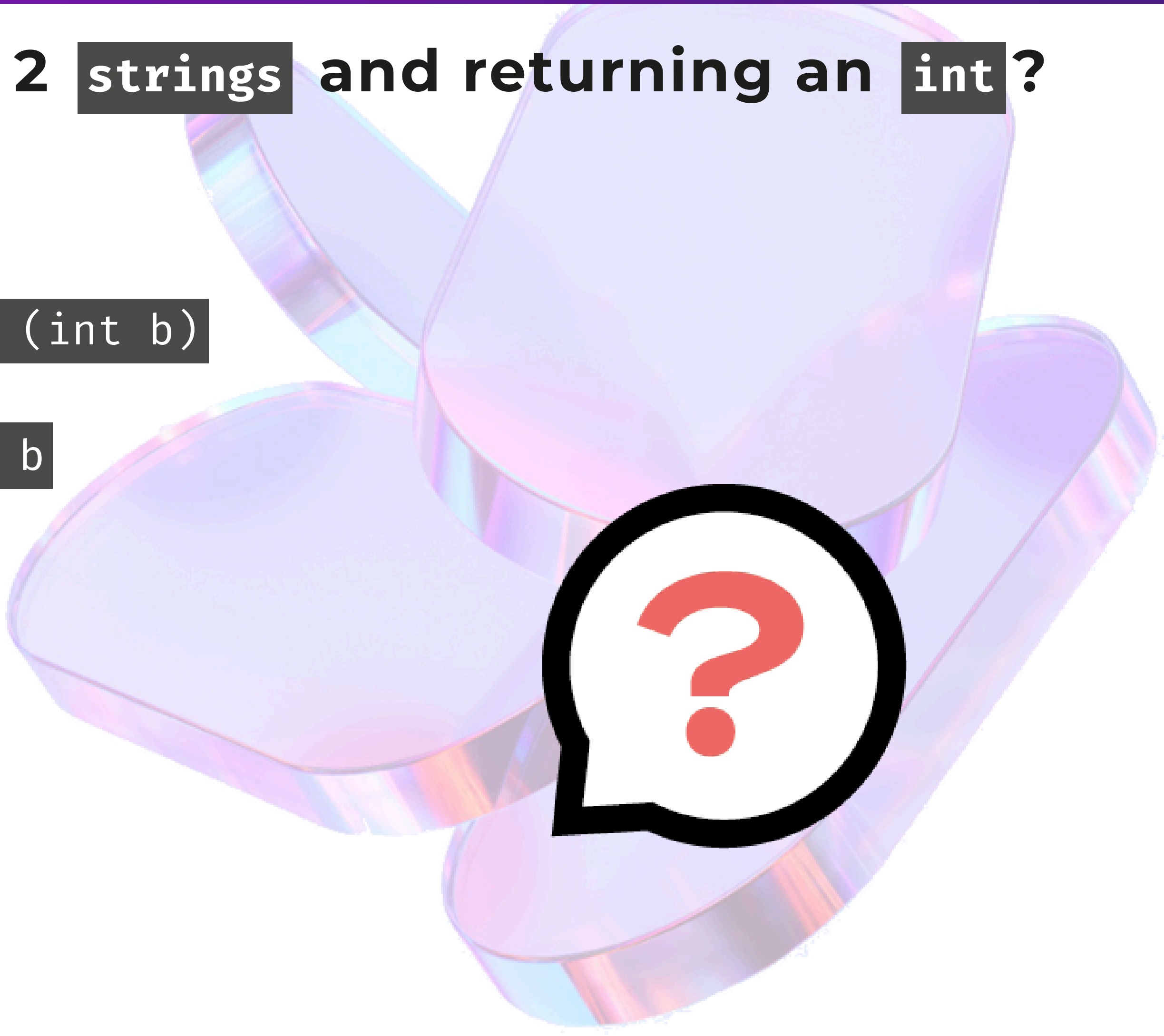
How to write an `add` function taking 2 `strings` and returning an `int`?

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''



Answer 2

How to write an `add` function taking 2 `strings` and returning an `int`?

A. `let add a b = a + b` ❌

| Wrong type inferred for `a` and `b` : `int`

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

| The type of `a` and `b` must be specified.

| They must be converted to `int`.

| The `int` return type can be inferred.

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

| Here, `+` does string concat

Question 3

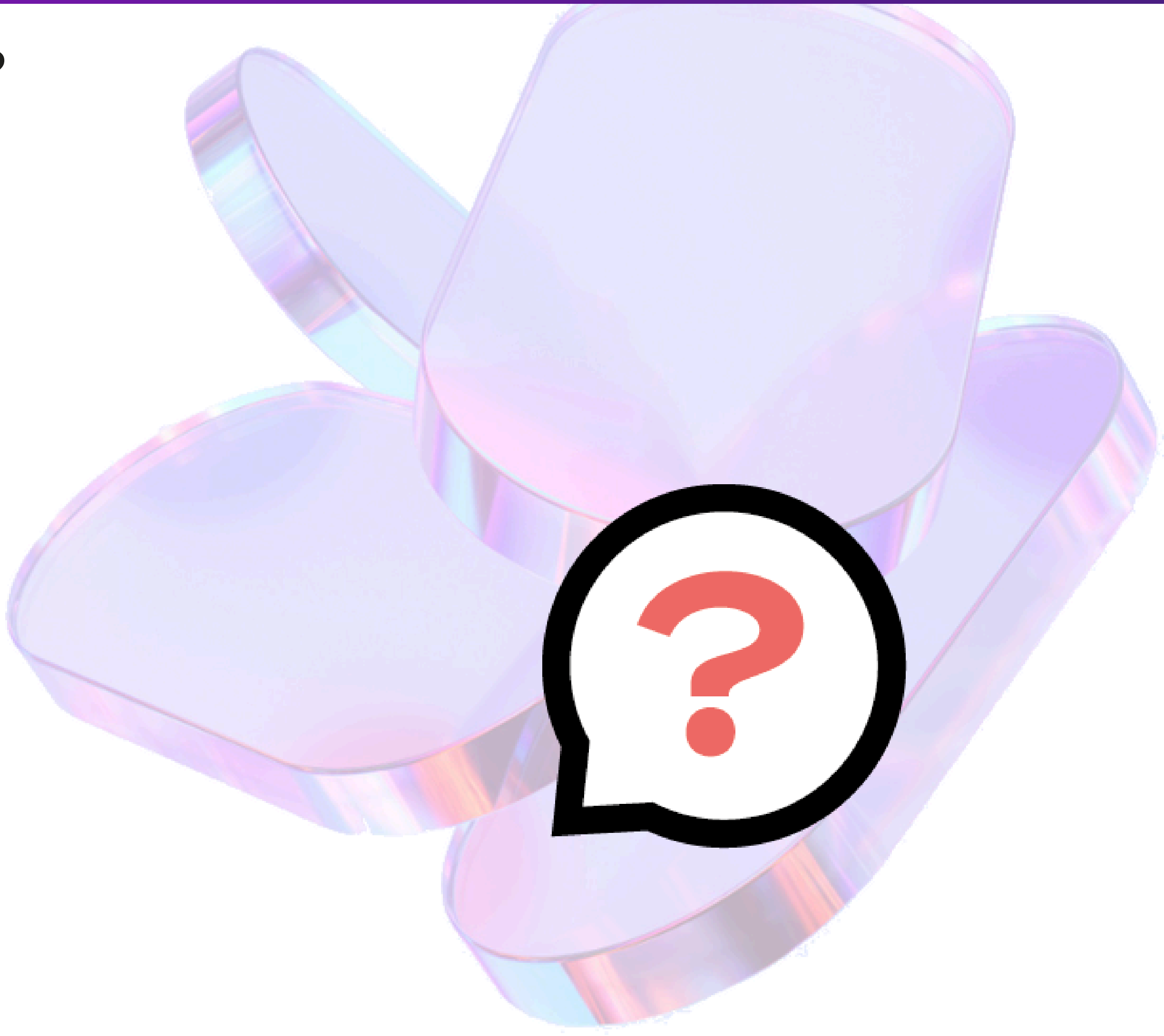
What does this code: `add >> multiply`?

A. Create a pipeline

B. Define a named function

C. Compose 2 functions

⏱️ 10''



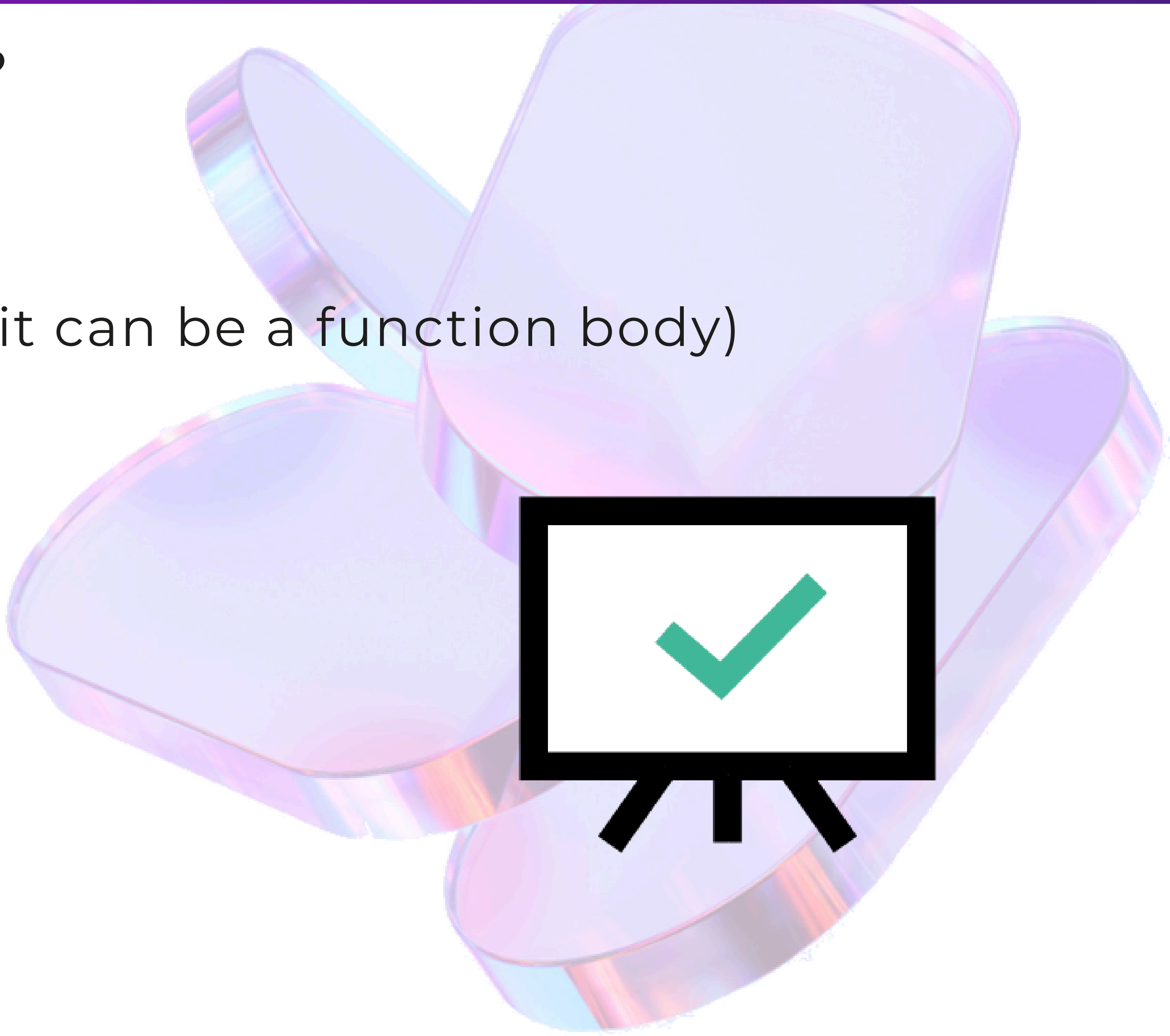
Answer 3

What does this code: `add >> multiply`?

A. Create a pipeline ❌

B. Define a named function ❌ (although it can be a function body)

C. Compose 2 functions ✅



Question 4

Find the name of these functions from `FSharp.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’’ 💡 Tips: These may be operators.

Answer 4

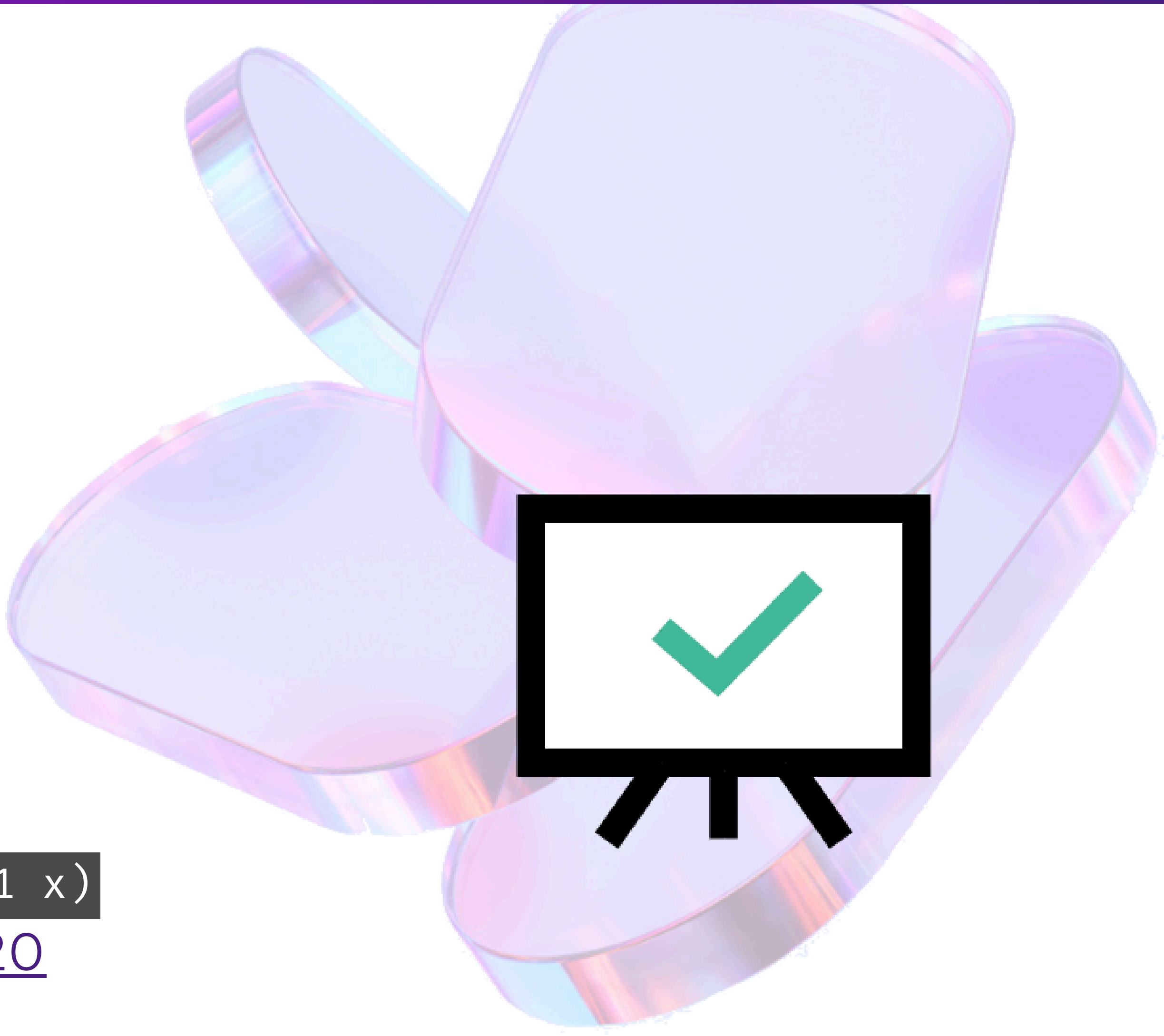
A. `let inline ignore _ = ()`
→ **Ignore** : prim-types.fs#L459

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



Q5. Describe functions from signature

number + type of parameters, return type

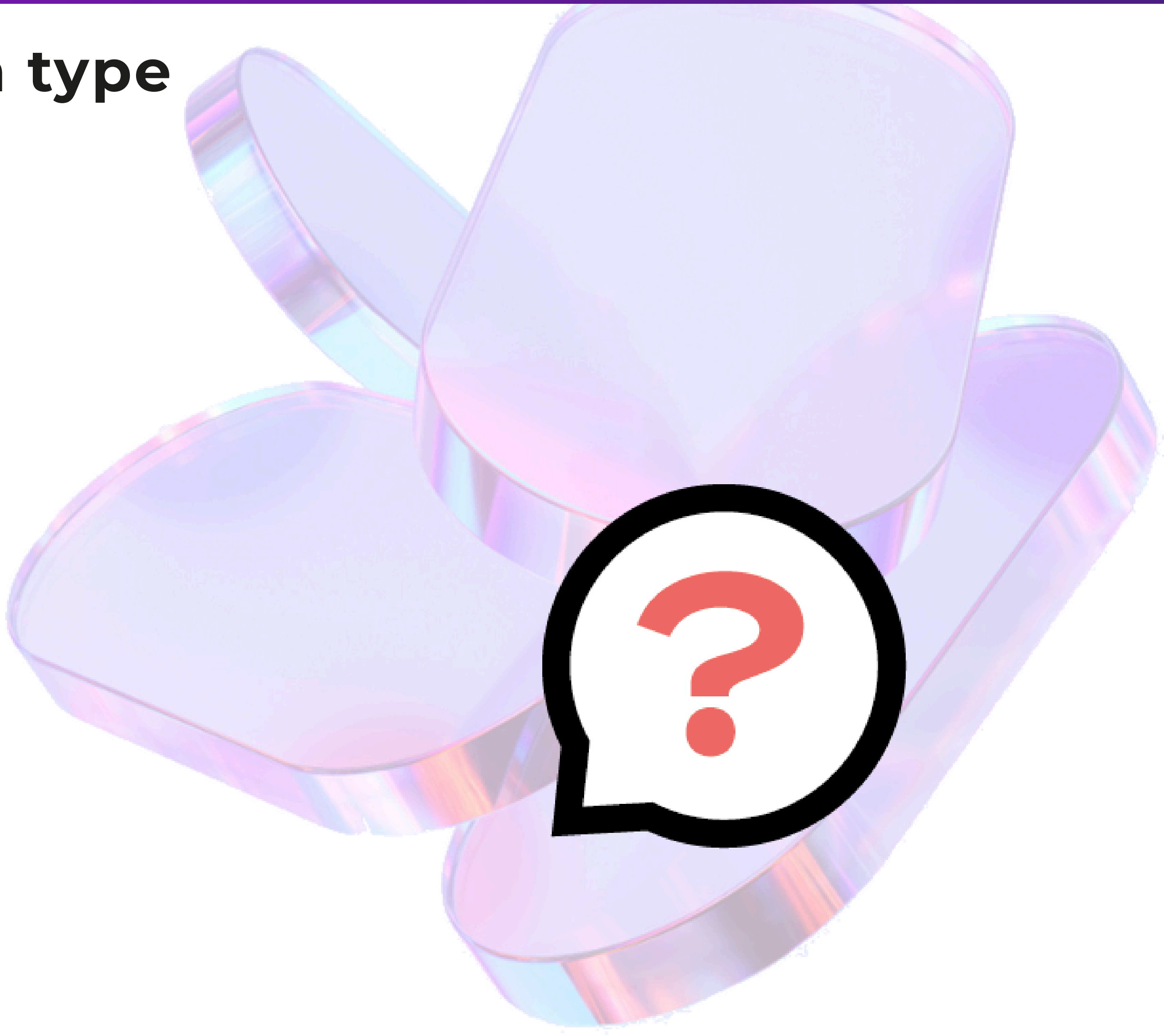
A. `int → unit`

B. `unit → int`

C. `string → string → string`

D. `('T → bool) → 'T list → 'T list`

⏱ 60’’



Answer 5. Describe functions

A. `int → unit`
1 parameter: `int` - no return value

B. `unit → int`
no parameter - return a `int`

C. `string → string → string`
2 parameters: `string` - return a `string`

D. `('T → bool) → 'T list → 'T list`
2 parameters: a predicate and a list - returns a list
→ `filter` function

Question 6. Signature of `h` ?

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

A. `int → int`

B. `int → string`

C. `int → int → string`

D. `int → int → int`

⏱ 30''

Answer 6. Signature of `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` can be written `let h x y = g (f x) y`

Addendum Q6.

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

This question was difficult...
... to illustrate the **misuse** of `>>`

→ **Tips**: Avoid compose functions having different arities!
(`f` has 1 parameter, `g` has 2).

Question 7. What value returns `f 2`?

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

A. `1`

B. `3`

C. `-1`

⏱ 10’’

Answer 7. What value returns `f 2`?

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

C. `-1` ❗

➔ Indeed, `f` can be written: `let f x = 1 - x`

➔ Counter-intuitive: we expect f to decrement by 1.

➔ Such a function can be written:

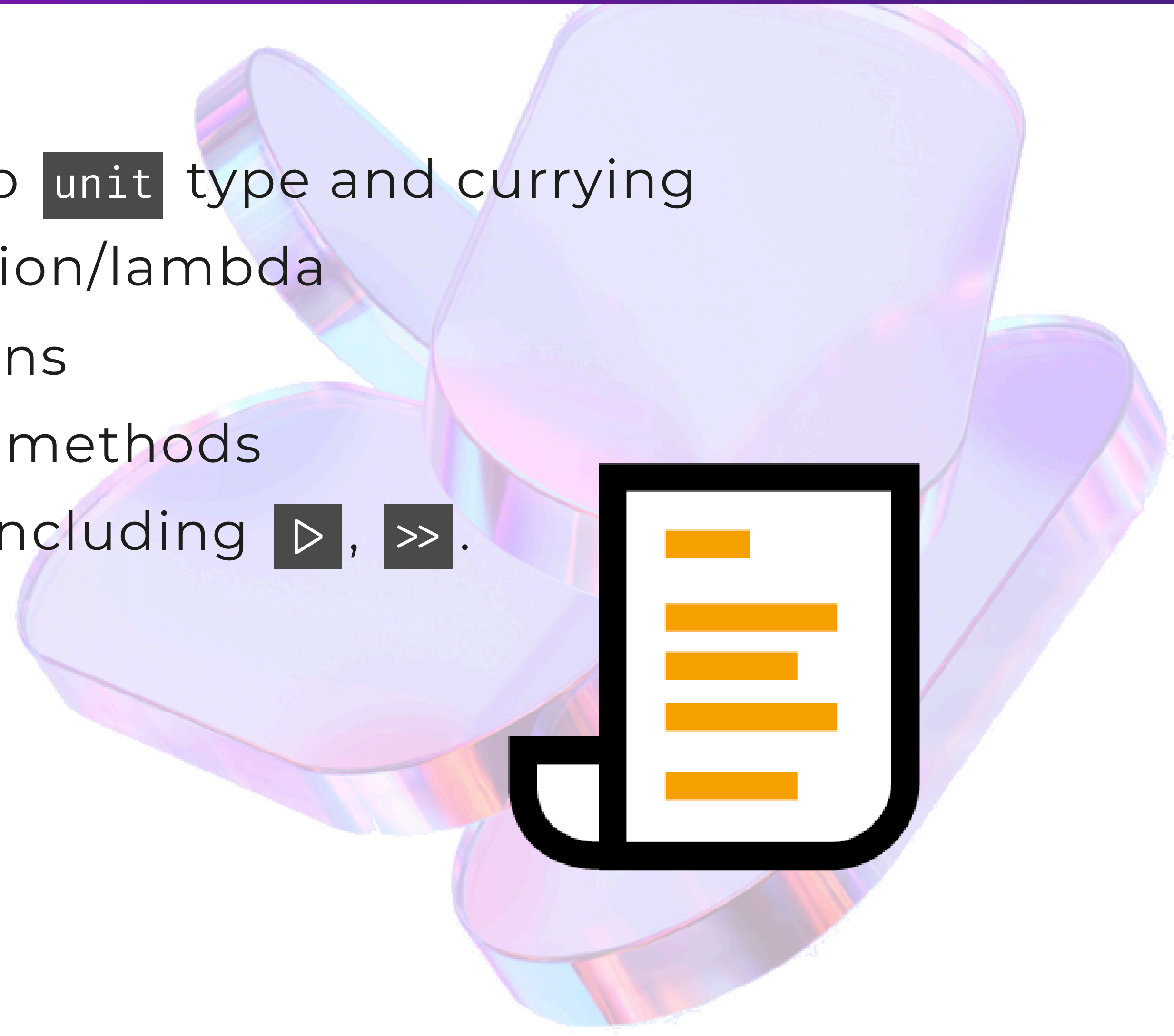
➔ `let f = (+) -1` *(`+` is commutative, `-` not)*

➔ `let f x = x - 1`

6. Wrap up

We've seen

- ➔ Signature with arrow notation
- ➔ Universal signature `T → U` thanks to `unit` type and currying
- ➔ Generic function, anonymous function/lambda
- ➔ Recursive and *tail recursion* functions
- ➔ Differences between functions and methods
- ➔ Standard functions and operators, including `▷`, `>>`.
- ➔ Overloading or creating operators
- ➔ Point-free notation
- ➔ Interoperability with BCL



A lot

- ➔ It's a lot, just for functions!
- ➔ But they are a corner stone in F♯.

Thanks 🙏