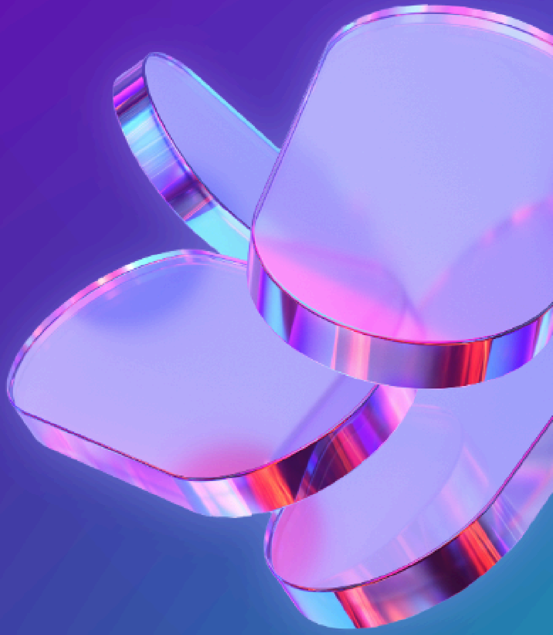
F♯ Training 📎

Computation Expressions (CE)

2025 July

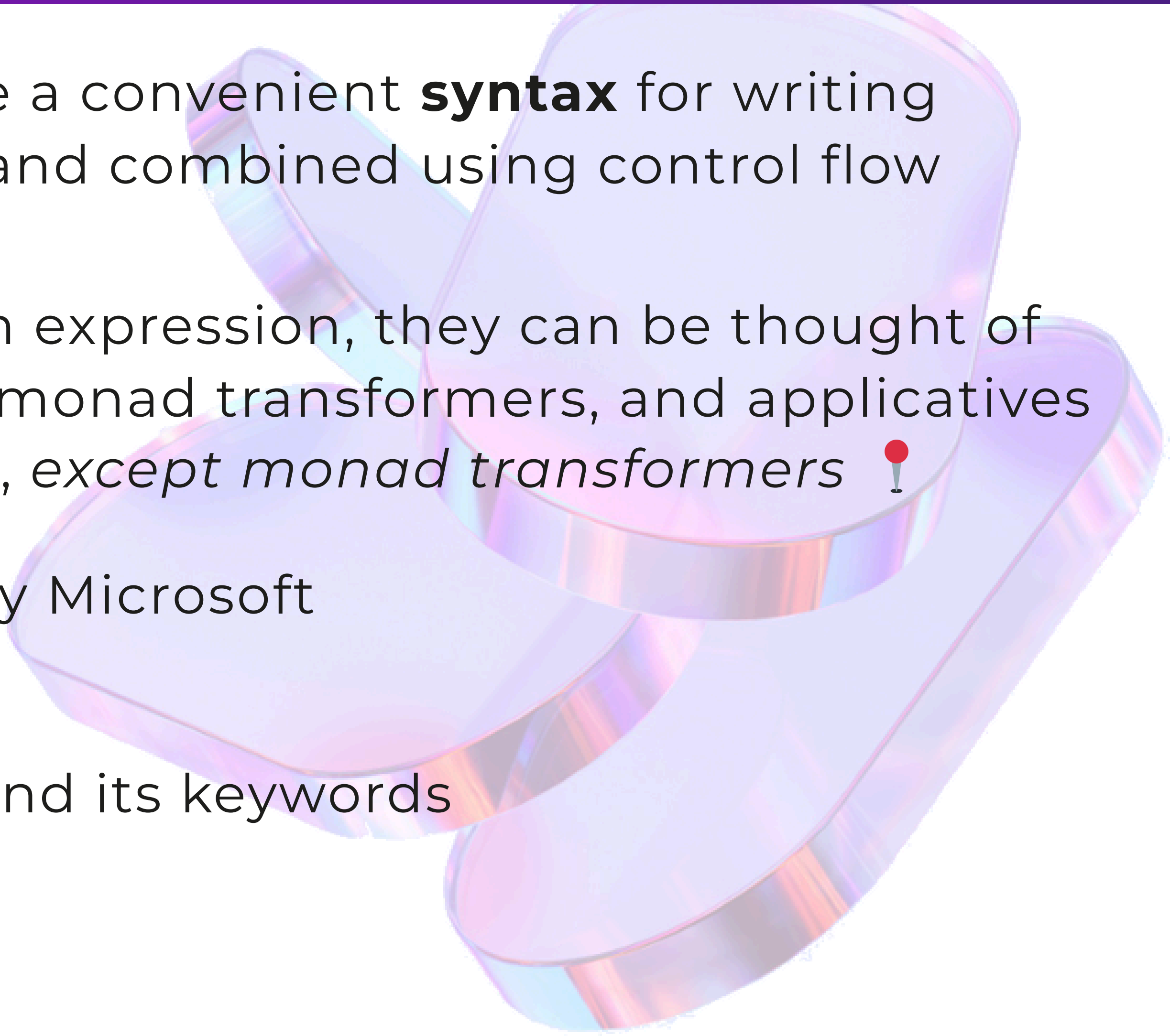


d-edge

Table of contents

1. Intro

Presentation



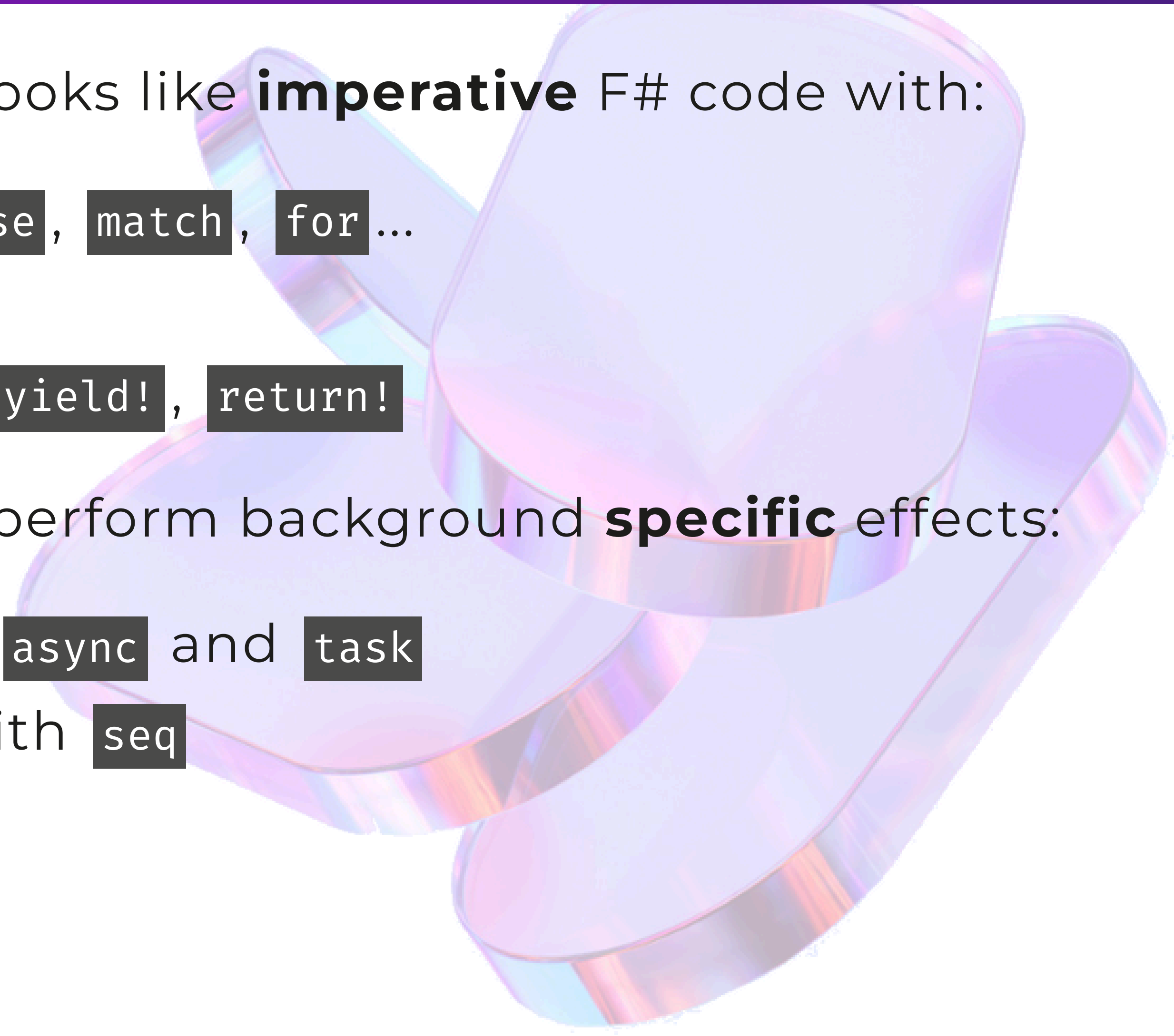
1. Computation expressions in F# provide a convenient **syntax** for writing computations that can be sequenced and combined using control flow constructs and bindings.
2. Depending on the kind of computation expression, they can be thought of as a way to express monads, monoids, monad transformers, and applicatives → **Functional patterns** seen previously, *except monad transformers* 📍

🔗 Learn F# - Computation Expressions, by Microsoft

Built-in CEs: `async` and `task`, `seq`, `query`
→ Easy to use, once we know the syntax and its keywords

We can write our own CE too
→ More challenging!

Syntax



CE = block like `myCE { body }` where `body` looks like **imperative** F# code with:

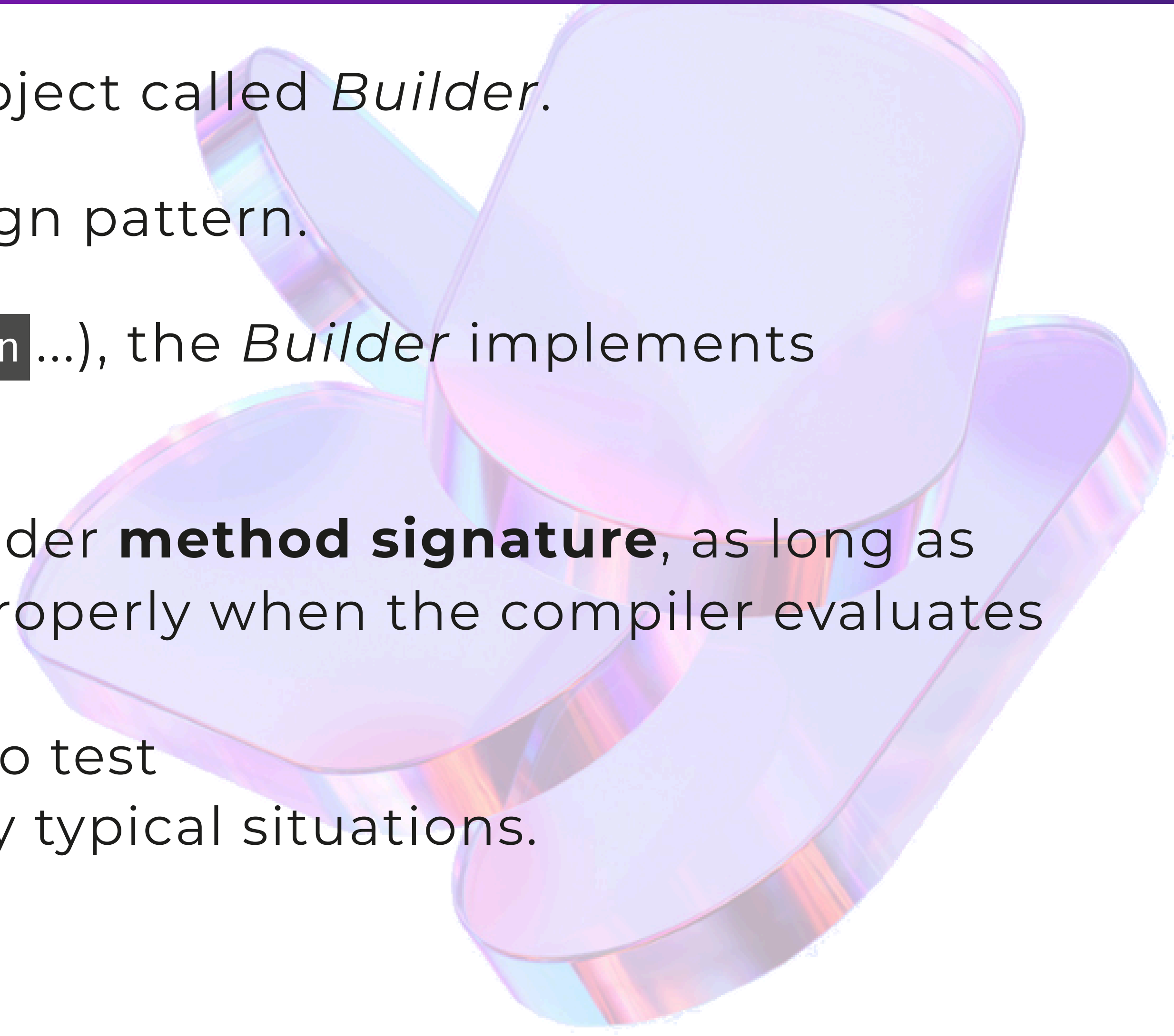
- regular keywords: `let`, `do`, `if`/`then`/`else`, `match`, `for`...
- dedicated keywords: `yield`, `return`
- "banged" keywords: `let!`, `do!`, `match!`, `yield!`, `return!`

These keywords hide a “ **machinery** ” to perform background **specific** effects:

- Asynchronous computations like with `async` and `task`
- State management: e.g. a sequence with `seq`
- Absence of value with `option` CE
- ...

2. Builder

Builder



A *computation expression* relies on an object called *Builder*.

⚠️ This is not exactly the *Builder* OO design pattern.

For each supported **keyword** (`let!`, `return`...), the *Builder* implements one or more related **methods**.

☝️ Compiler accepts **flexibility** in the builder **method signature**, as long as the methods can be **chained together** properly when the compiler evaluates the CE on the **caller side.**
→ ✅ Versatile, ⚠️ Difficult to design and to test
→ Given method signatures illustrate only typical situations.

Builder example: `logger {}`

Need: log the intermediate values of a calculation

```
// First version
let log value = printfn $"{value}"

let loggedCalc =
    let x = 42
    log x  // ❶
    let y = 43
    log y  // ❶
    let z = x + y
    log z  // ❶
    z
```

Issues ⚠️

① *Verbose*: the `log x` calls interfere with reading

② *Error prone*: easy to forget to log a value,
or to log the wrong variable after a bad copy-paste-update…

Builder example: `logger {}` (2)

💡 V2: make logs implicit in a CE by implementing a custom `let!` / `Bind()` :

```fsharp
type LoggingBuilder() =
    let log value = printfn $"{value}"; value
    member _.Bind(x, f) = x ▷ log ▷ f
    member _.Return(x) = x

let logger = LoggingBuilder()

//---

let loggedCalc = logger {
    let! x = 42      // 👉 Implicitly perform `log x`
    let! y = 43      // 👉                    `log y`
    let! z = x + y   // 👉                    `log z`
    return z
}
```

Builder example: `logger {}` (3)

The 3 consecutive `let!` are desugared into 3 **nested** calls to `Bind` with:

- 1st argument: the right side of the `let!` (e.g. `42` with `let! x = 42`)
- 2nd argument: a lambda taking the variable defined at the left side of the `let!` (e.g. `x`) and returning the whole expression below the `let!` until the `}`

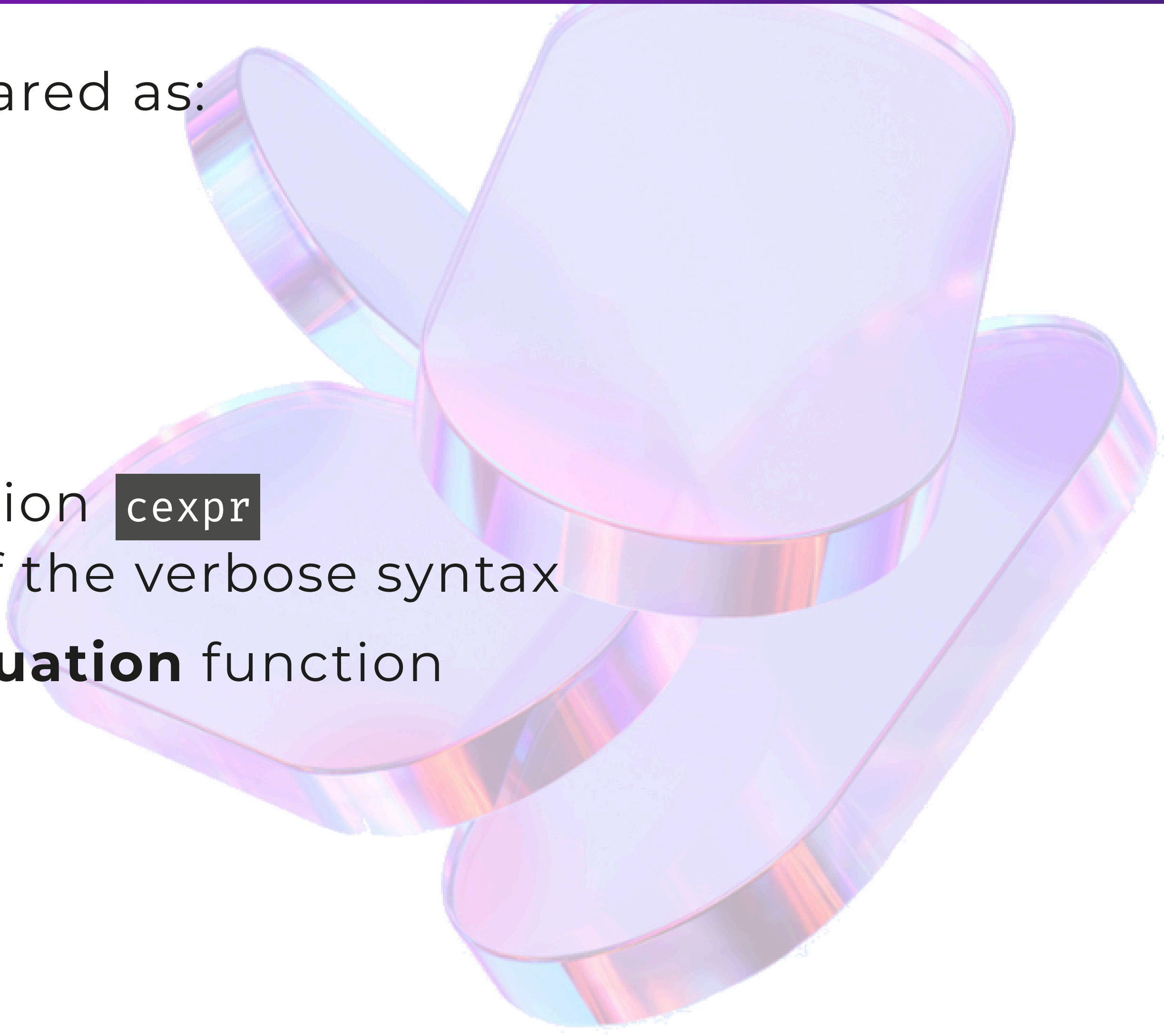
```
// let! x = 42
logger.Bind(42, (fun x →
    // let! y = 43
    logger.Bind(43, (fun y →
        // let! z = x + y
        logger.Bind(x + y, (fun z →
            logger.Return z)
        ))
    ))
)
```

Builder - Bind *vs* let!

`logger { let! var = expr in cexpr }` is desugared as:
`logger.Bind(expr, fun var → cexpr)`

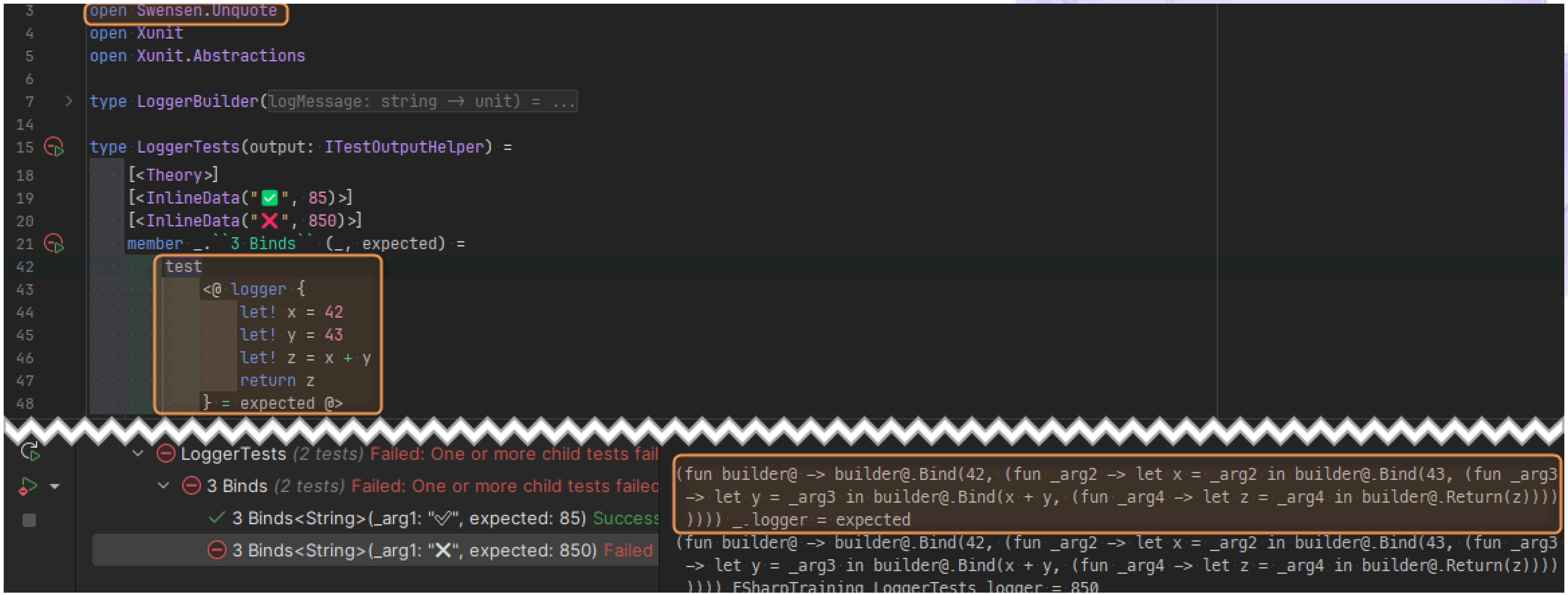
👉 **Key points:**

- `var` and `expr` appear in reverse order
- `var` is used in the rest of the computation `cexpr`
  → highlighted using the `in` keyword of the verbose syntax
- the lambda `fun var → cexpr` is a **continuation** function



CE desugaring: tips 💡

I found a simple way to desugar a computation expression:
→ Write a failing unit test and use Unquote - 🔗 Example



```fsharp
open Swensen.Unquote
open Xunit
open Xunit.Abstractions

type LoggerBuilder(logMessage: string -> unit) = ...

type LoggerTests(output: ITestOutputHelper) =
    [<Theory>]
    [<InlineData("✅", 85)>]
    [<InlineData("❌", 850)>]
    member _.``3 Binds`` (_, expected) =
        test
            <@ logger {
                let! x = 42
                let! y = 43
                let! z = x + y
                return z
            } = expected @>
```

```
LoggerTests (2 tests) Failed: One or more child tests fail
  3 Binds (2 tests) Failed: One or more child tests failed
    ✓ 3 Binds<String>(_arg1: "✅", expected: 85) Success
    3 Binds<String>(_arg1: "❌", expected: 850) Failed
```

```
(fun builder@ -> builder@.Bind(42, (fun _arg2 -> let x = _arg2 in builder@.Bind(43, (fun _arg3
 -> let y = _arg3 in builder@.Bind(x + y, (fun _arg4 -> let z = _arg4 in builder@.Return(z))))
 )))) _.logger = expected
(fun builder@ -> builder@.Bind(42, (fun _arg2 -> let x = _arg2 in builder@.Bind(43, (fun _arg3
 -> let y = _arg3 in builder@.Bind(x + y, (fun _arg4 -> let z = _arg4 in builder@.Return(z))))
 )))) FSharpTraining.LoggerTests.logger = 850
```

Builder constructor parameter

The builder can be constructed with additional parameters.
→ The CE syntax allows us to pass these arguments when using the CE:

```fsharp
type LoggingBuilder(prefix: string) =
    let log value = printfn $"{prefix}{value}"; value
    member _.Bind(x, f) = x ▷ log ▷ f
    member _.Return(x) = x

let logger prefix = LoggingBuilder(prefix)

// ---

let loggedCalc = logger "[Debug] " {
    let! x = 42      // 👉 Output "[Debug] 42"
    let! y = 43      // 👉 Output "[Debug] 43"
    let! z = x + y   // 👉 Output "[Debug] 85"
    return z
}
```

Builder example: `option {}`

Need: successively try to find in maps by identifiers
→ Steps:

1. `roomRateId` in `policyCodesByRoomRate` map → `policyCode`
2. `policyCode` in `policyTypesByCode` map → `policyType`
3. `policyCode` and `policyType` → `result`

```
// 1: with match expressions → nesting!
match policyCodesByRoomRate.TryFind(roomRateId) with
| None -> None
| Some policyCode ->
    match policyTypesByCode.TryFind(policyCode) with
    | None -> None
    | Some policyType -> Some(buildResult policyCode policyType)
```

Builder example: `option {}` (2)

```
// 2: with Option module helpers → terser but harder to read
policyCodesByRoomRate.TryFind(roomRateId)
▷ Option.bind (fun policyCode → policyCode, policyTypesByCode.TryFind(policyCode))
▷ Option.map (fun (policyCode, policyType) → buildResult policyCode policyType)
```

Builder example: `option {}` (3)

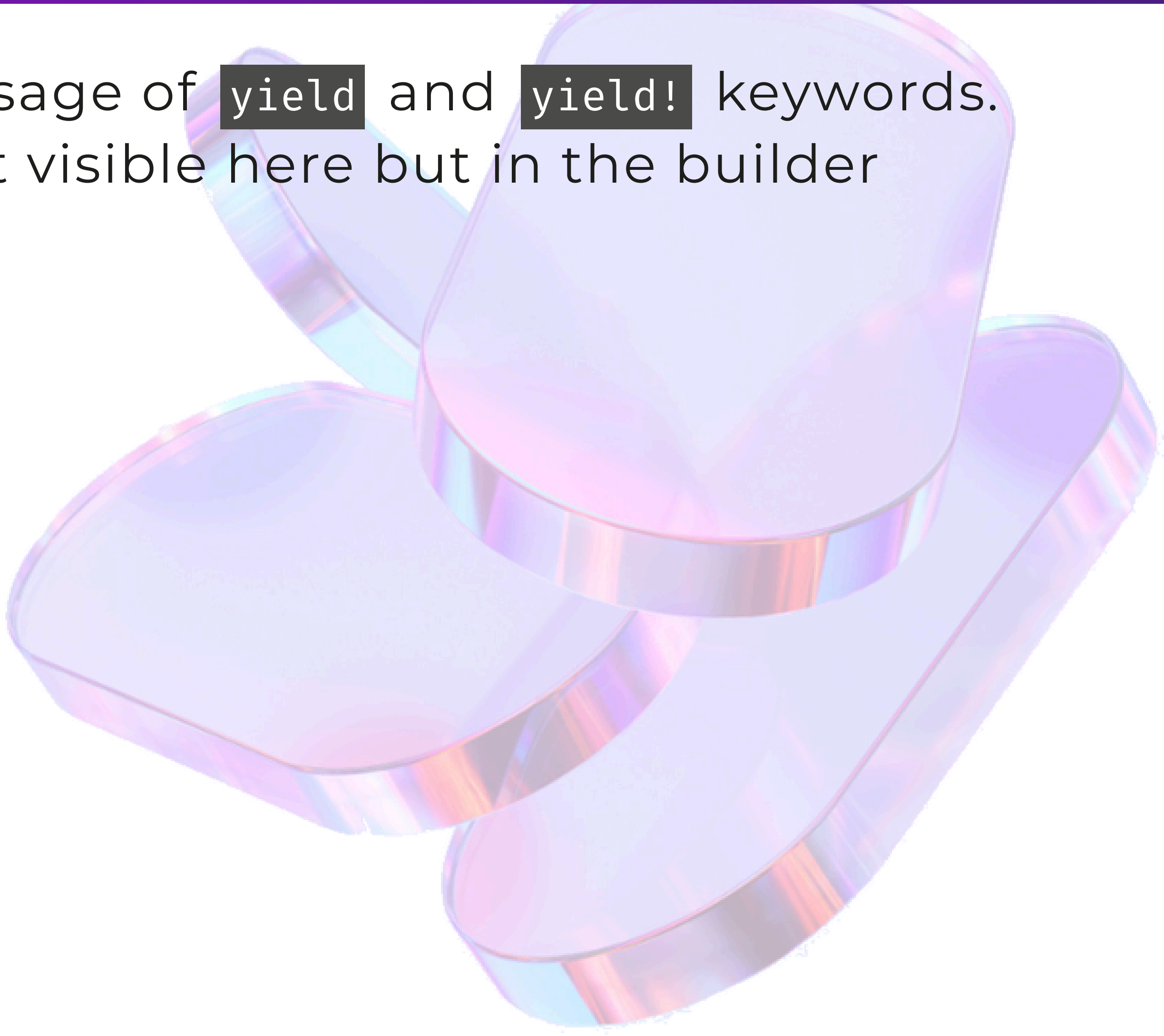
```fsharp
// 3: with an option CE → both terse and readable 🎉

type OptionBuilder() =
    member _.Bind(x, f) = x ▷ Option.bind f
    member _.Return(x) = Some x

let option = OptionBuilder()

// ---

option {
    let! policyCode = policyCodesByRoomRate.TryFind(roomRateId)
    let! policyType = policyTypesByCode.TryFind(policyCode)
    return buildResult policyCode policyType
}
```

3. CE monoidal

CE monoidal

A monoidal CE can be identified by the usage of `yield` and `yield!` keywords.
→ The relationship with the monoid is not visible here but in the builder methods:

- `+` operation → `Combine` method
- `e` neutral element → `Zero` method



CE monoidal builder methods

```
// Method     | Signature                                  | CE syntax supported
    Yield     : T → M<T>                                   ; yield x
    YieldFrom : M<T> → M<T>                                ; yield! xs
    Zero      : unit → M<T>                                ; if // without `else`   // Monoid neutral element
    Combine   : M<T> * M<T> → M<T>                                                  // Monoid + operation
    Delay     : (unit → M<T>) → Delayed<T>                 ; // always required with Combine
    For       : seq<T> * (T → M<U>) → M<U>                 ; for i in seq do yield ... ; for i = 0 to n do yield ...
                                  (* or *)  seq<M<U>>

// Other additional methods
    Run       : Delayed<T> → M<T>
    While     : (unit → bool) * Delayed<T> → M<T> ; while cond do yield...
    TryWith   : M<T> → (exn → M<T>) → M<T>        ; try/with
    TryFinally: Delayed<T> * (unit → unit) → M<T> ; try/finally
```

1. We use the generic type notation `M<T>`, like we did for functional patterns.
2. Same for `Delayed<T>`, presented later 📍

CE monoidal *vs* comprehension

- Similar syntax from caller perspective
- Distinct overlapping concepts

Comprehension is the concise, declarative syntax to build collections with ranges `start..end` and control flow keywords `if`, `for`, `while`...

Minimal set of methods expected for each:

- Monoidal CE: `Yield`, `Combine`, `Zero`
- Comprehension: `For`, `Yield`

CE monoidal example: `multiplication {}` (1)

Let's build a CE that multiplies the integers yielded in the computation body:
→ CE type: `M<T> = int` · Monoid operation = `*` · Neutral element = `1`

```fsharp
type MultiplicationBuilder() =
    member _.Zero() = 1
    member _.Yield(x) = x
    member _.Combine(x, y) = x * y
    member _.Delay(f) = f () // eager evaluation

    member m.For(xs, f) =
        (m.Zero(), xs)
        |> Seq.fold (fun res x -> m.Combine(res, f x))

let multiplication = MultiplicationBuilder()

let shouldBe10 = multiplication { 5; 2 } // 👈 Implicit `yield`s // = 5 * 2
let factorialOf5 = multiplication { for i in 2..5 -> i } // 2 * 3 * 4 * 5
```

CE monoidal example: `multiplication {}` (2)

Desugared `multiplication { 5; 2 }`:

```
// Original
let shouldBe10 =
    multiplication.Delay(fun () →
        multiplication.Combine(
            multiplication.Yield(5),
            multiplication.Delay(fun () →
                multiplication.Yield(2)
            )
        )
    )

// Simplified (without Delay)
let shouldBe10 =
    multiplication.Combine(
        multiplication.Yield(5),
        multiplication.Yield(2)
    )
```

CE monoidal example: `multiplication {}` (3)

Desugared `multiplication { for i in 2..5 -> i }`:

```fsharp
// Original
let factorialOf5 =
    multiplication.Delay (fun () ->
        multiplication.For({2..5}, (fun _arg2 ->
            let i = _arg2 in multiplication.Yield(i))
        )
    )

// Simplified
let factorialOf5 =
    multiplication.For({2..5}, (fun i -> multiplication.Yield(i)))
```

CE monoidal `Delayed<'t>` type (1/3)

`Delayed<'t>` represents a delayed computation and is used in these methods:

- `Delay` defines the CE `Delayed<'t>` as its return type
- `Combine`, `Run`, `While` and `Tryfinally` used it as input parameter

```
Delay      : thunk: (unit → M<T>) → Delayed<T>
Combine    : M<T> * Delayed<T> → M<T>
Run        : Delayed<T> → M<T>
While      : predicate: (unit → bool) * Delayed<T> → M<T>
TryFinally : Delayed<T> * finalizer: (unit → unit) → M<T>
```

- `Delay` is called with the current expression (of type `M<T>`) before each method taking a `Delayed<T>` argument
- `Delayed<T>` is internal to the CE → given `Delayed<T>` ≠ `M<T>`, `Run` is called at the end of the chain to get back the `M<T>`

CE monoidal `Delayed<'t>` type (2/3)

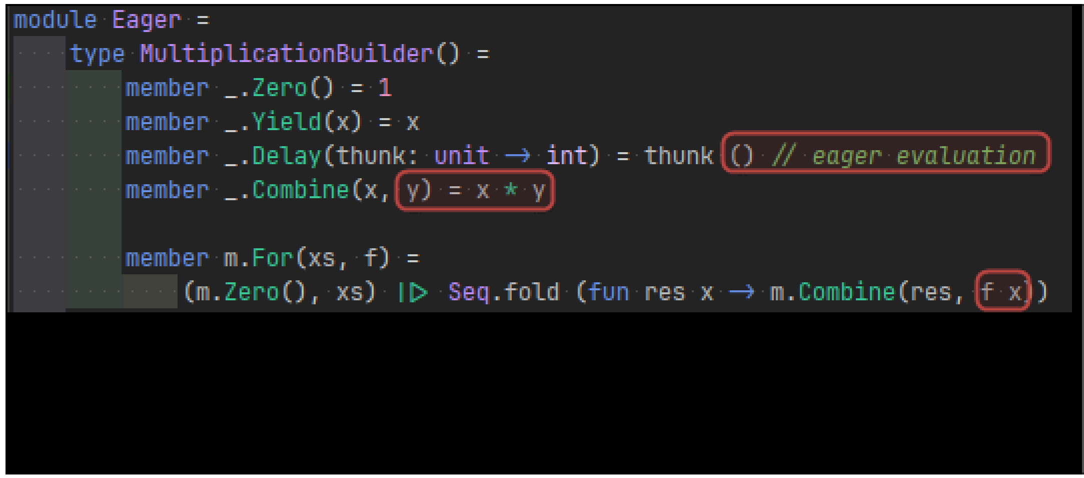
👉 Enables to implement **laziness and short-circuiting** at the CE level.

Example: lazy `multiplication {}` with `Combine` optimized when `x = 0`

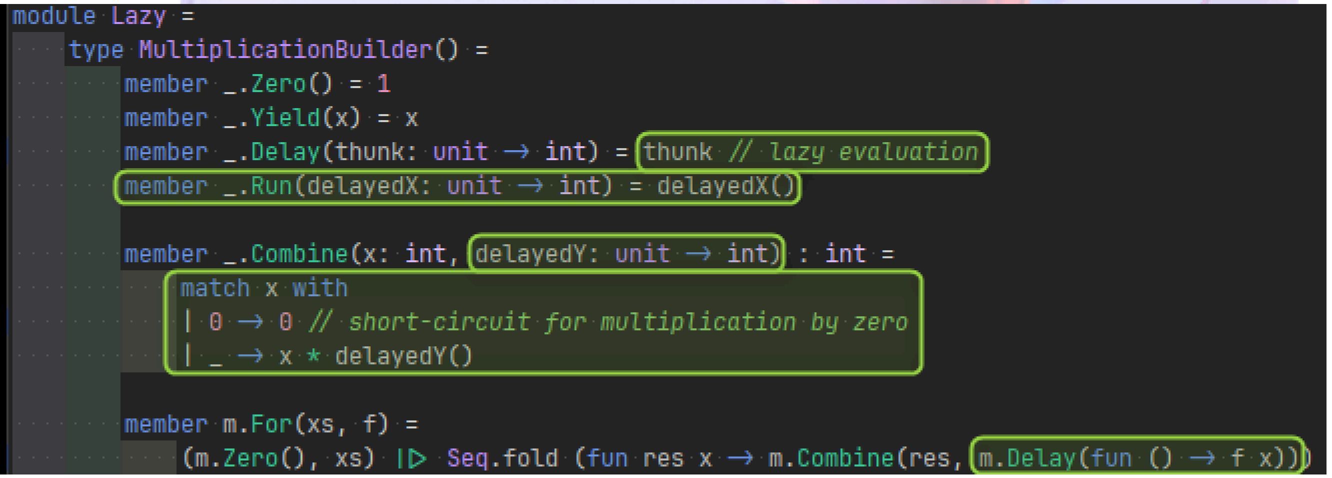
```fsharp
type MultiplicationBuilder() =
    member _.Zero() = 1
    member _.Yield(x) = x
    member _.Delay(thunk: unit → int) = thunk // Lazy evaluation
    member _.Run(delayedX: unit → int) = delayedX ()

    member _.Combine(x: int, delayedY: unit → int) : int =
        match x with
        | 0 → 0 // Short-circuit for multiplication by zero
        | _ → x * delayedY ()

    member m.For(xs, f) =
        (m.Zero(), xs) |▷ Seq.fold (fun res x → m.Combine(res, fun () → f x))
```

CE monoidal `Delayed<'t>` type (3/3)

Difference	Eager	Lazy
`Delay` return type	`int`	`unit → int`
`Run`	Omitted	Required to get back an `int`
`Combine` 2nd parameter	`int`	`unit → int`
`For` calling `Delay`	Omitted	Explicit but not required



```
module Eager =
    type MultiplicationBuilder() =
        member _.Zero() = 1
        member _.Yield(x) = x
        member _.Delay(thunk: unit → int) = thunk () // eager evaluation
        member _.Combine(x, y) = x * y

        member m.For(xs, f) =
            (m.Zero(), xs) |▷ Seq.fold (fun res x → m.Combine(res, f x))
```



```
module Lazy =
    type MultiplicationBuilder() =
        member _.Zero() = 1
        member _.Yield(x) = x
        member _.Delay(thunk: unit → int) = thunk // lazy evaluation
        member _.Run(delayedX: unit → int) = delayedX()

        member _.Combine(x: int, delayedY: unit → int) : int =
            match x with
            | 0 → 0 // short-circuit for multiplication by zero
            | _ → x * delayedY()

        member m.For(xs, f) =
            (m.Zero(), xs) |▷ Seq.fold (fun res x → m.Combine(res, m.Delay(fun () → f x)))
```

CE monoidal to generate a collection (1/4)

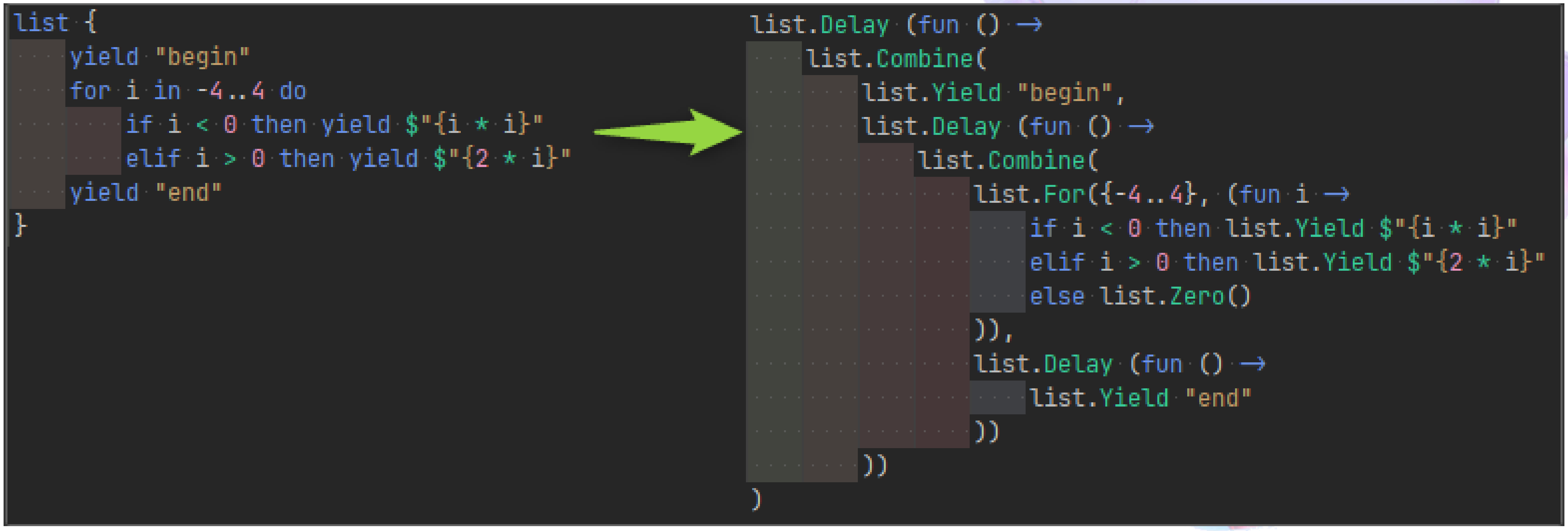
`multiplication {}` returns an `int` from possibility multiple yielded ints.
≠ `seq {}` returns a `'t seq` from any number of yielded value of type `t`
→ It's a more common use case for a monoidal CE.

💡 Let's build a `list {}` monoidal CE!

```fsharp
type ListBuilder() =
    member _.Zero() = [] // List.empty
    member _.Yield(x) = [x] // List.singleton
    member _.YieldFrom(xs) = xs
    member _.Delay(thunk: unit → 't list) = thunk () // eager evaluation
    member _.Combine(xs, ys) = xs @ ys // List.append
    member _.For(xs, f) = xs ▷ Seq.collect f ▷ Seq.toList

let list = ListBuilder()
```

CE monoidal to generate a collection (2/4)

Let's test the CE to generate the list `[begin; 16; 9; 4; 1; 2; 4; 6; 8; end]`
(Desugared code simplified)

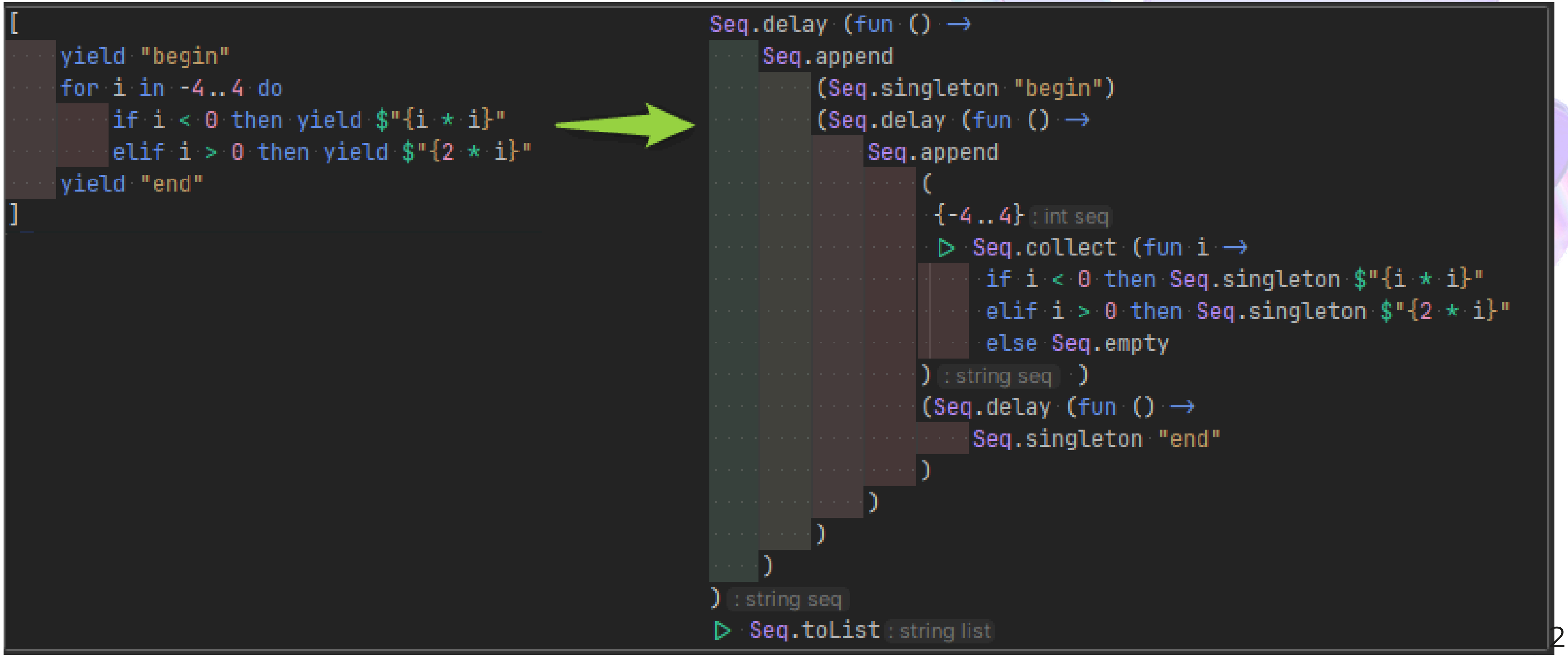


```fsharp
list {
    yield "begin"
    for i in -4..4 do
        if i < 0 then yield $"{i * i}"
        elif i > 0 then yield $"{2 * i}"
    yield "end"
}
```

```fsharp
list.Delay (fun () ->
    list.Combine(
        list.Yield "begin",
        list.Delay (fun () ->
            list.Combine(
                list.For({-4..4}, (fun i ->
                    if i < 0 then list.Yield $"{i * i}"
                    elif i > 0 then list.Yield $"{2 * i}"
                    else list.Zero()
                )),
                list.Delay (fun () ->
                    list.Yield "end"
                ))
        ))
)
```

CE monoidal to generate a collection (3/4)

Comparison with the same expression in a regular list:



```
[
    yield "begin"
    for i in -4..4 do
        if i < 0 then yield $"{i * i}"
        elif i > 0 then yield $"{2 * i}"
    yield "end"
]
```

```
Seq.delay (fun () ->
    Seq.append
        (Seq.singleton "begin")
        (Seq.delay (fun () ->
            Seq.append
                (
                 {-4..4} : int seq
                 |> Seq.collect (fun i ->
                    if i < 0 then Seq.singleton $"{i * i}"
                    elif i > 0 then Seq.singleton $"{2 * i}"
                    else Seq.empty
                ) : string seq )
                (Seq.delay (fun () ->
                    Seq.singleton "end"
                )
            )
        )
    )
) : string seq
|> Seq.toList : string list
```

CE monoidal to generate a collection (4/4)

`list { expr }` *vs* `[ expr ]`:

- `[ expr ]` uses a hidden `seq` all through the computation and ends with a `toList`
- All methods are inlined:

Method	`list { expr }`	`[ expr ]`
`Combine`	`xs @ ys ⇒ List.append`	`Seq.append`
`Yield`	`[x]       ⇒ List.singleton`	`Seq.singleton`
`Zero`	`[]        ⇒ List.empty`	`Seq.empty`
`For`	`Seq.collect` & `Seq.toList`	`Seq.collect`

4. CE monadic

CE monadic

A monadic CE can be identified by the usage of `let!` and `return` keywords, revealing the monadic `bind` and `return` operations.

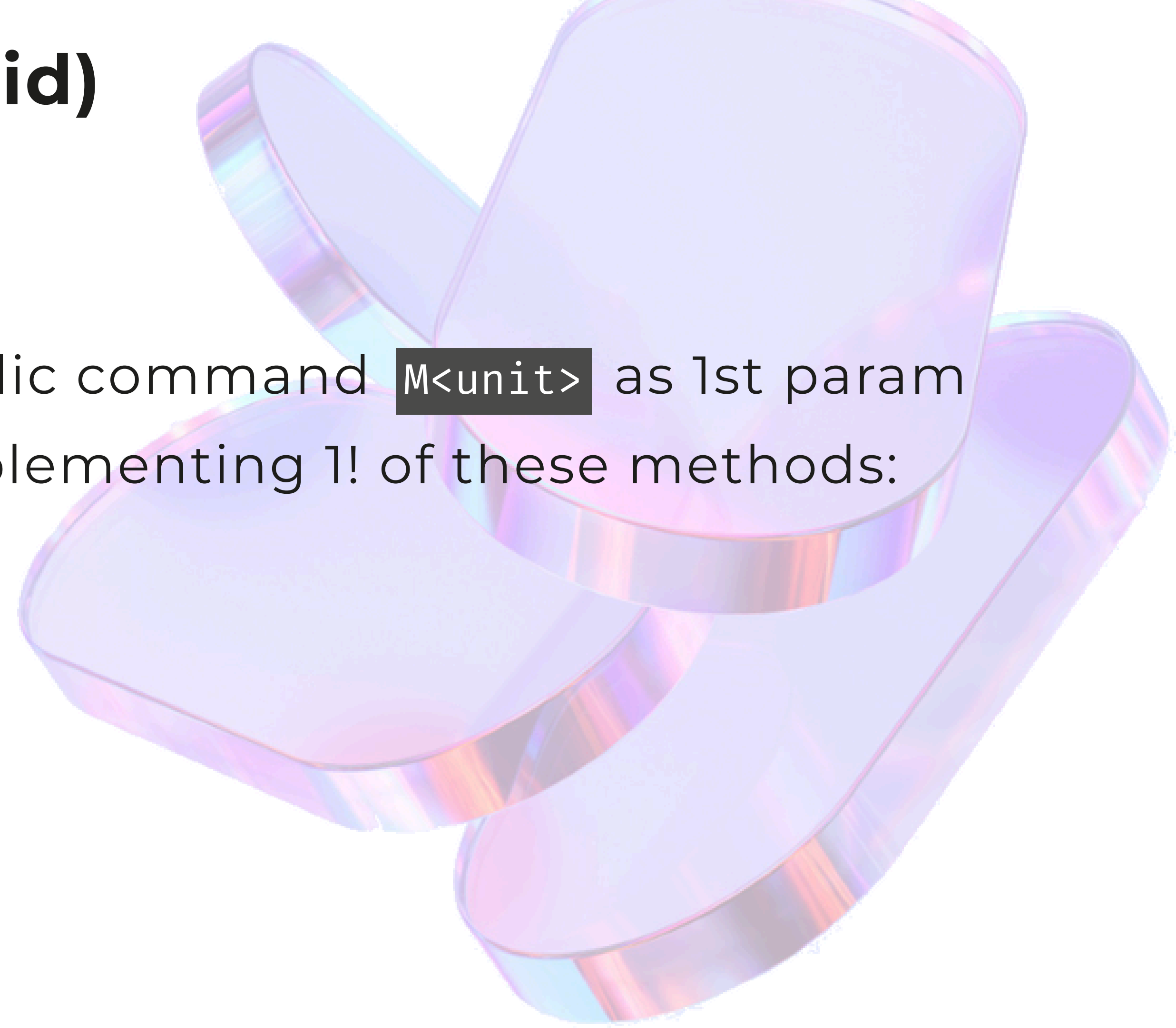
Behind the scene, builders of these CE should/can implement these methods:

```
// Method     | Signature                                    | CE syntax supported
    Bind      : M<T> * (T → M<U>) → M<U>                    ; let! x = xs in ...
                (* when T = unit *)                          ; do! command
    Return    : T → M<T>                                     ; return x
    ReturnFrom: M<T> → M<T>                                  ; return!

// Additional methods
    Zero      : unit → M<T>                                  ; if // without `else` // Typically `unit → M<unit>`
    Combine   : M<unit> * M<T> → M<T>                        ; e1; e2  // e.g. one loop followed by another one
    TryWith   : M<T> → (exn → M<T>) → M<T>                   ; try/with
    TryFinally: M<T> * (unit → M<unit>) → M<T>               ; try/finally
    While     : (unit → bool) * (unit → M<unit>) → M<unit> ; while cond do command ()
    For       : seq<T> * (T → M<unit>) → M<unit>             ; for i in xs do command i ; for i = 0 to n do command i
    Using     : T * (T → M<U>) → M<U> when T :> IDisposable ; use! x = xs in ...
```

CE monadic *vs* CE monoidal (1/2)

`Return` (monad) *vs* `Yield` (monoid)

- Same signature: `T → M<T>`
- A series of `return` is not expected
  → Monadic `Combine` takes only a monadic command `M<unit>` as 1st param
- CE enforces appropriate syntax by implementing 1! of these methods:
  - `seq {}` allows `yield` but not `return`
  - `async {}`: vice versa



CE monadic *vs* CE monoidal (2/2)

`For` and `While`

Method	CE	Signature
`For`	Monoidal	`seq<T> * (T → M<U>)     → M<U> or seq<M<U>>`
	Monadic	`seq<T> * (T → M<unit>) → M<unit>`
`While`	Monoidal	`(unit → bool) * Delayed<T>          → M<T>`
	Monadic	`(unit → bool) * (unit → M<unit>) → M<unit>`

👉 ≠ use cases:

- Monoidal: Comprehension syntax
- Monadic: Series of effectful commands

CE monadic and delayed

Like monoidal CE, monadic CE can use a `Delayed<'t>` type.
→ Impacts on the method signatures:

```
Delay       : thunk: (unit → M<T>) → Delayed<T>
Run         : Delayed<T> → M<T>
Combine     : M<unit> * Delayed<T> → M<T>
While       : predicate: (unit → bool) * Delayed<unit> → M<unit>
TryFinally  : Delayed<T> * finalizer: (unit → unit) → M<T>
TryWith     : Delayed<T> * handler: (exn → unit) → M<T>
```

CE monadic examples (1/2)

☝️ The initial CE studied—`logger {}` and `option {}`—was monadic.

Other example: `result {}` CE handling:

- `if` w/o `else` → `Zero`
- `do! command` → `Combine`
- lazy evaluation → `Delay`, `Run`

```fsharp
type ResultBuilder() =
    member _.Bind(rx, f) = rx |> Result.bind f
    member _.Return(x) = Ok x

    member m.Zero() = m.Return(()) // = Ok ()
    member m.Combine(cmd: Result<unit, _>, delayedRY) = m.Bind(cmd, (fun () -> delayedRY ()))
    member _.Delay(thunk: unit -> Result<_, _>) = thunk
    member _.Run(delayedRX: unit -> Result<_, _>) = delayedRX ()

let result = ResultBuilder()
```

CE monadic examples (2/2)

```fsharp
let rollDice =
    let random = Random(Guid.NewGuid().GetHashCode())
    fun () -> random.Next(1, 7)

let tryGetDice dice =
    result {
        if rollDice() <> dice then
            return! Error $"Not the expected dice {dice}."
    }

let atRoll n res =
    match res with
    | Ok x -> Ok x
    | Error msg -> Error $"{msg} at roll #{n}"

let tryGet421 () =
    result {
        do! (tryGetDice 4 |> atRoll 1)
        do! (tryGetDice 2 |> atRoll 2)
        do! (tryGetDice 1 |> atRoll 3)
    }
```

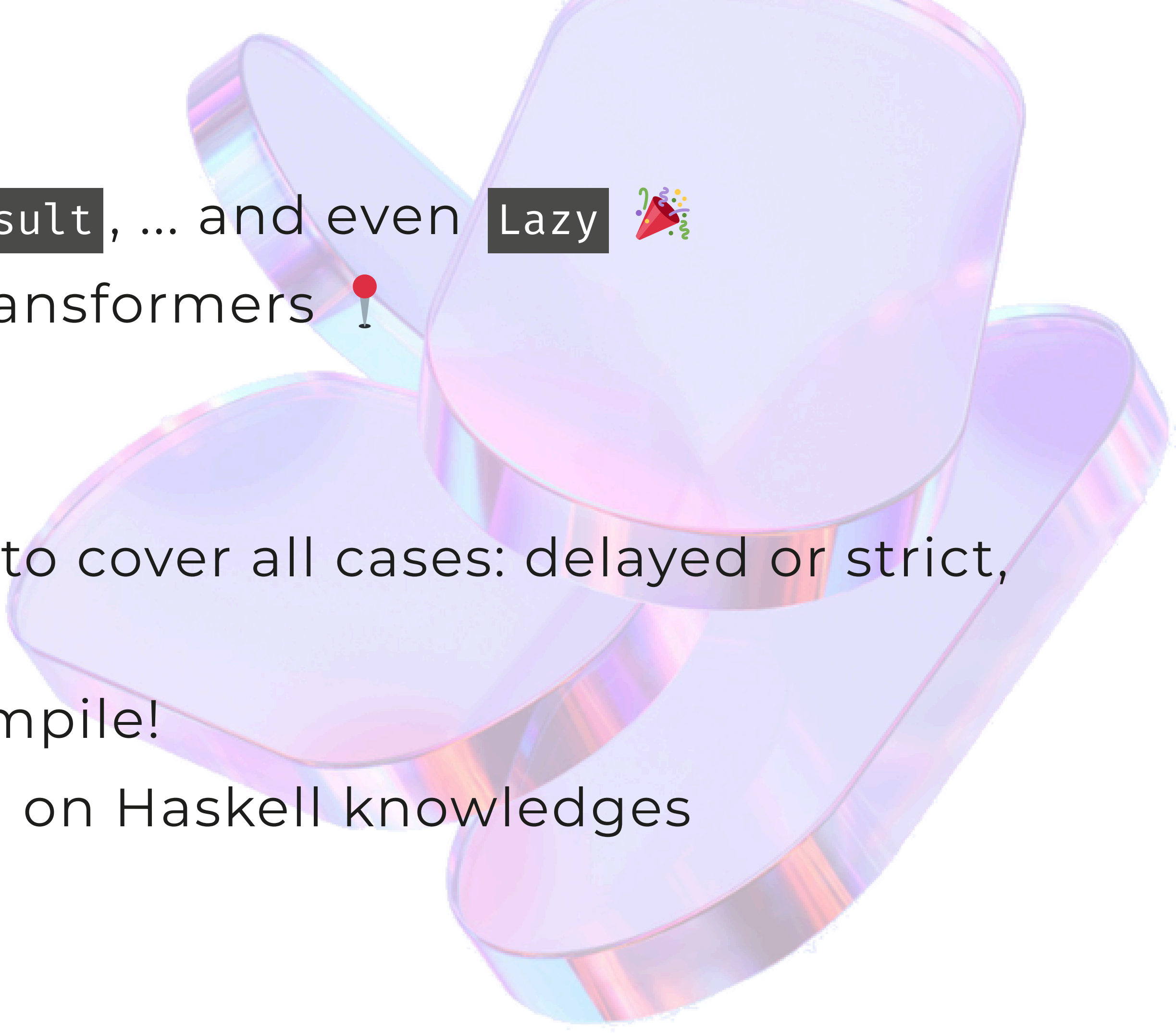
CE monadic: FSharpPlus `monad` CE

FSharpPlus provides a `monad` CE

- Works for all monadic types: `Option`, `Result`, ... and even `Lazy` 🎉
- Supports monad stacks with monad transformers 📍

⚠️ **Limits:**

- Confusing: the `monad` CE has 4 flavours to cover all cases: delayed or strict, embedded side-effects or not
- Based on SRTP: can be very long to compile!
- Documentation not exhaustive, relying on Haskell knowledges
- Very Haskell-oriented: not idiomatic F♯



Monad stack, monad transformers

A monad stack is a composition of different monads.
→ Example: `Async` + `Option`.

How to handle it?
→ Academic style *vs* idiomatic F♯

1. Academic style (with FSharpPlus)

Monad transformer (here `MaybeT`)
→ Extends `Async` to handle both effects
→ Resulting type: `MaybeT<Async<'t>>`

✅ reusable with other inner monad
❌ less easy to evaluate the resulting value
❌ not idiomatic

Monad stack, monad transformers (2)

2. Idiomatic style

Custom CE `asyncOption`, based on the `async` CE, handling `Async<Option<'t>>` type

```fsharp
type AsyncOption<'T> = Async<Option<'T>> // Convenient alias, not required

type AsyncOptionBuilder() =
    member _.Bind(aoX: AsyncOption<'a>, f: 'a → AsyncOption<'b>) : AsyncOption<'b> =
        async {
            match! aoX with
            | Some x → return! f x
            | None → return None
        }

    member _.Return(x: 'a) : AsyncOption<'a> =
        async { return Some x }
```

⚠️ Limits: not reusable, just copiable for `asyncResult` for instance

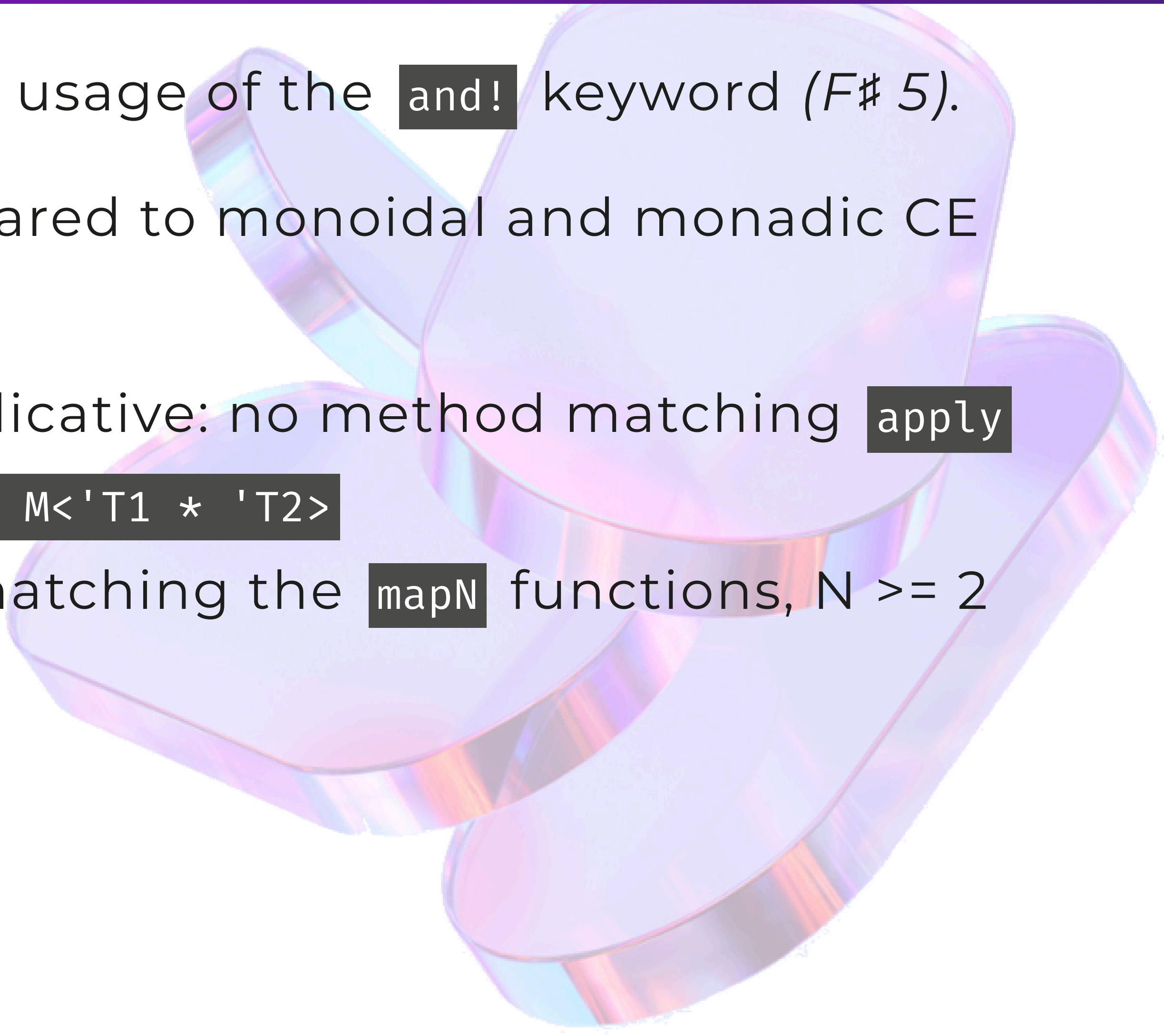
5. CE Applicative

CE Applicative

An applicative CE is revealed through the usage of the `and!` keyword *(F♯ 5)*.

An applicative CE builder is special compared to monoidal and monadic CE builders:

- Not following the definition of the applicative: no method matching `apply`
- Based on `MergeSources: M<'T1> * M<'T2> → M<'T1 * 'T2>`
- Fine tuning with `BindNReturn` method matching the `mapN` functions, N >= 2



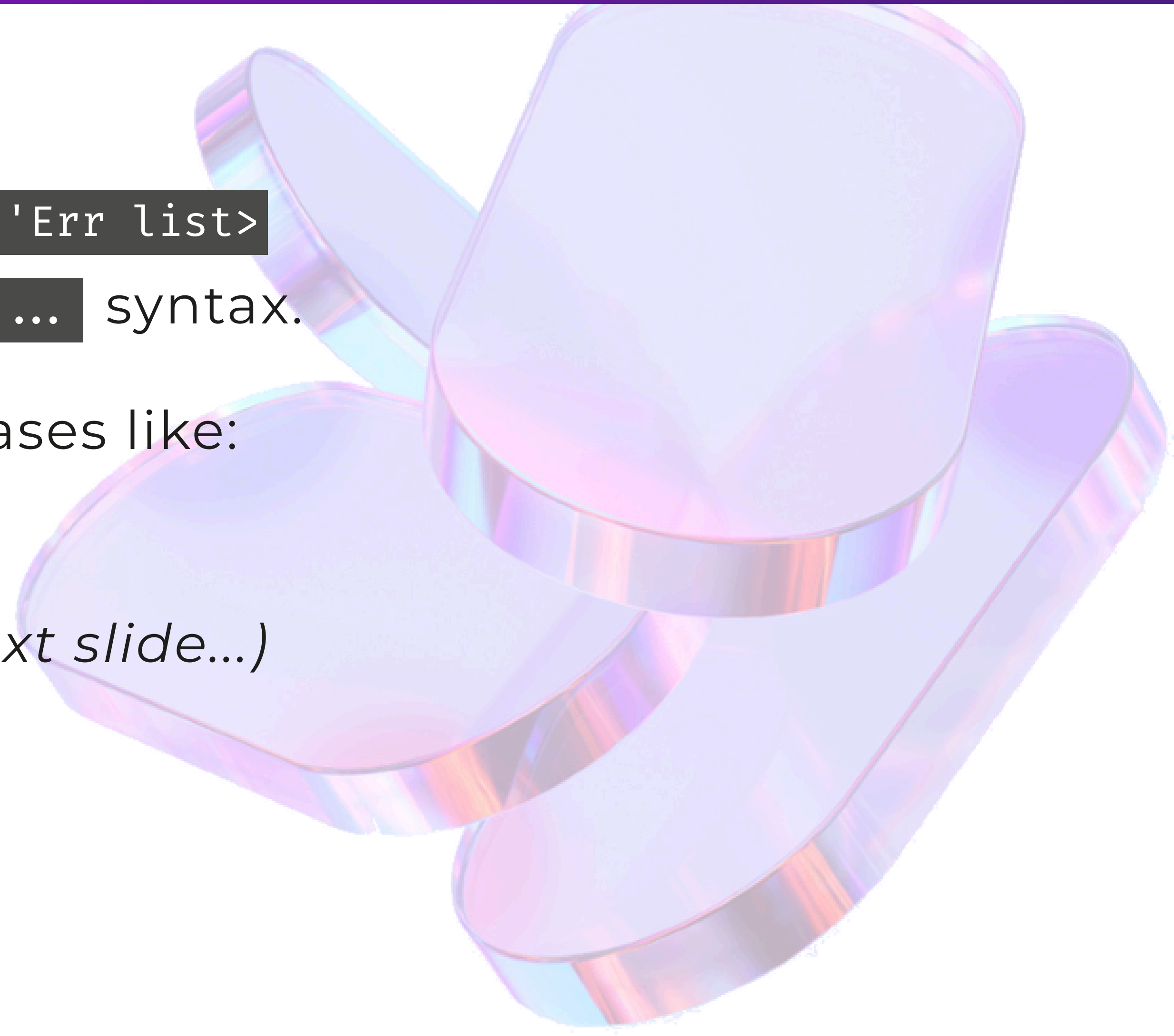
CE Applicative example

FsToolkit.ErrorHandling offers:

- Type `Validation<'Ok, 'Err>` ≡ `Result<'Ok, 'Err list>`
- CE `validation {}` supporting `let! ... and! ...` syntax.

Allows errors to be accumulated in use cases like:

- Parsing external inputs
- *Smart constructor (example on the next slide...)*



CE Applicative example (2)

```fsharp
#r "nuget: FSToolkit.ErrorHandling"
open FsToolkit.ErrorHandling

type [<Measure>] cm
type Customer = { Name: string; Height: int<cm> }

let validateHeight height =
    if height ⩽ 0<cm>
    then Error "Height must be positive"
    else Ok height

let validateName name =
    if System.String.IsNullOrWhiteSpace name
    then Error "Name can't be empty"
    else Ok name

module Customer =
    let tryCreate name height : Result<Customer, string list> =
        validation {
            let! validName = validateName name
            and! validHeight = validateHeight height
            return { Name = validName; Height = validHeight }
        }

let c1 = Customer.tryCreate "Bob" 180<cm>  // Ok { Name = "Bob"; Height = 180 }
let c2 = Customer.tryCreate "Bob" 0<cm> // Error ["Height must be positive"]
let c3 = Customer.tryCreate "" 0<cm>    // Error ["Name can't be empty"; "Height must be positive"]
```

6. Creating CEs

Types

The CE builder methods definition can involve not 2 but 3 types:

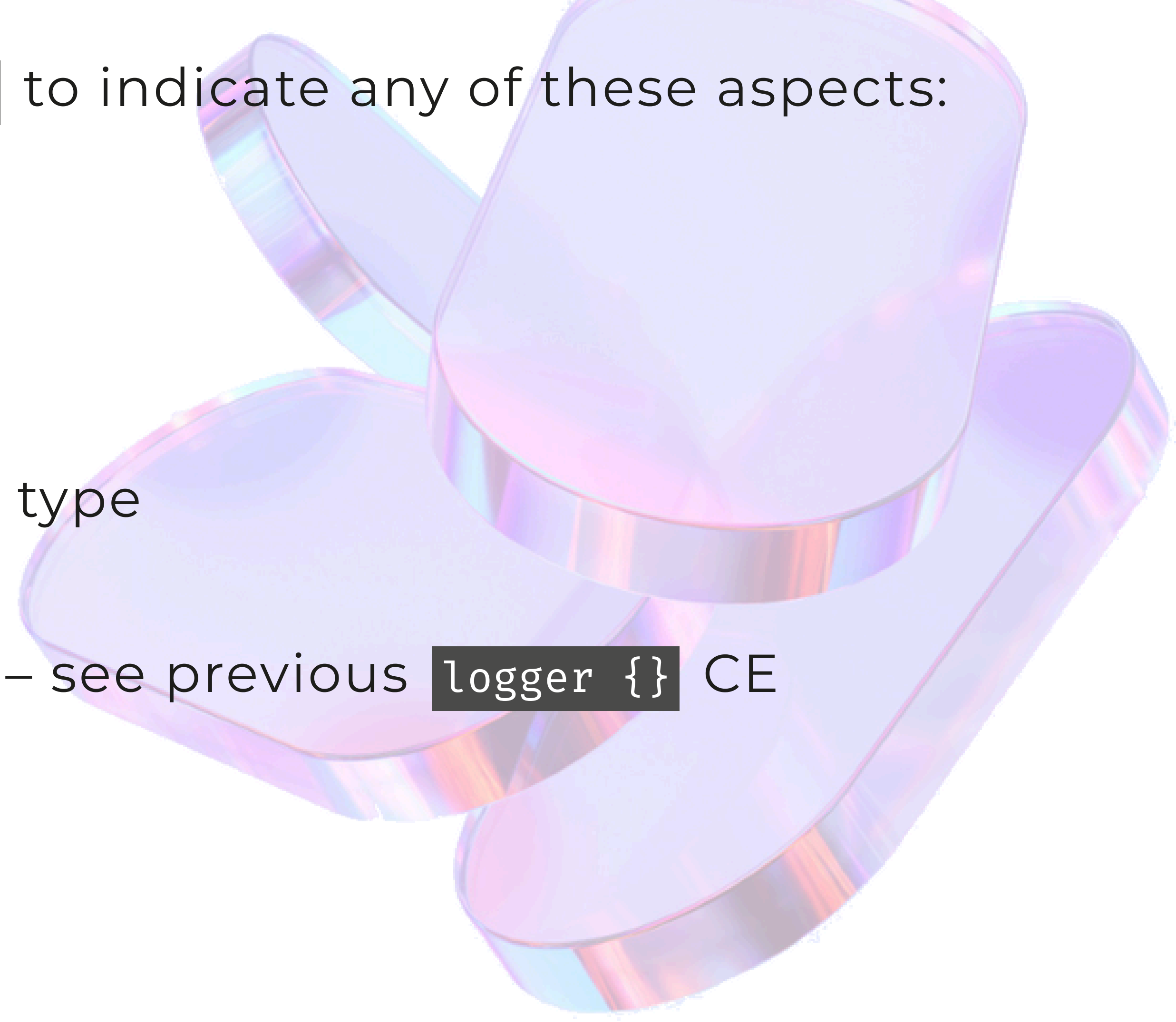
- The wrapper type `M<T>`
- The `Delayed<T>` type
- An `Internal<T>` type

`M<T>` wrapper type

☝️ We use the generic type notation `M<T>` to indicate any of these aspects: generic or container.

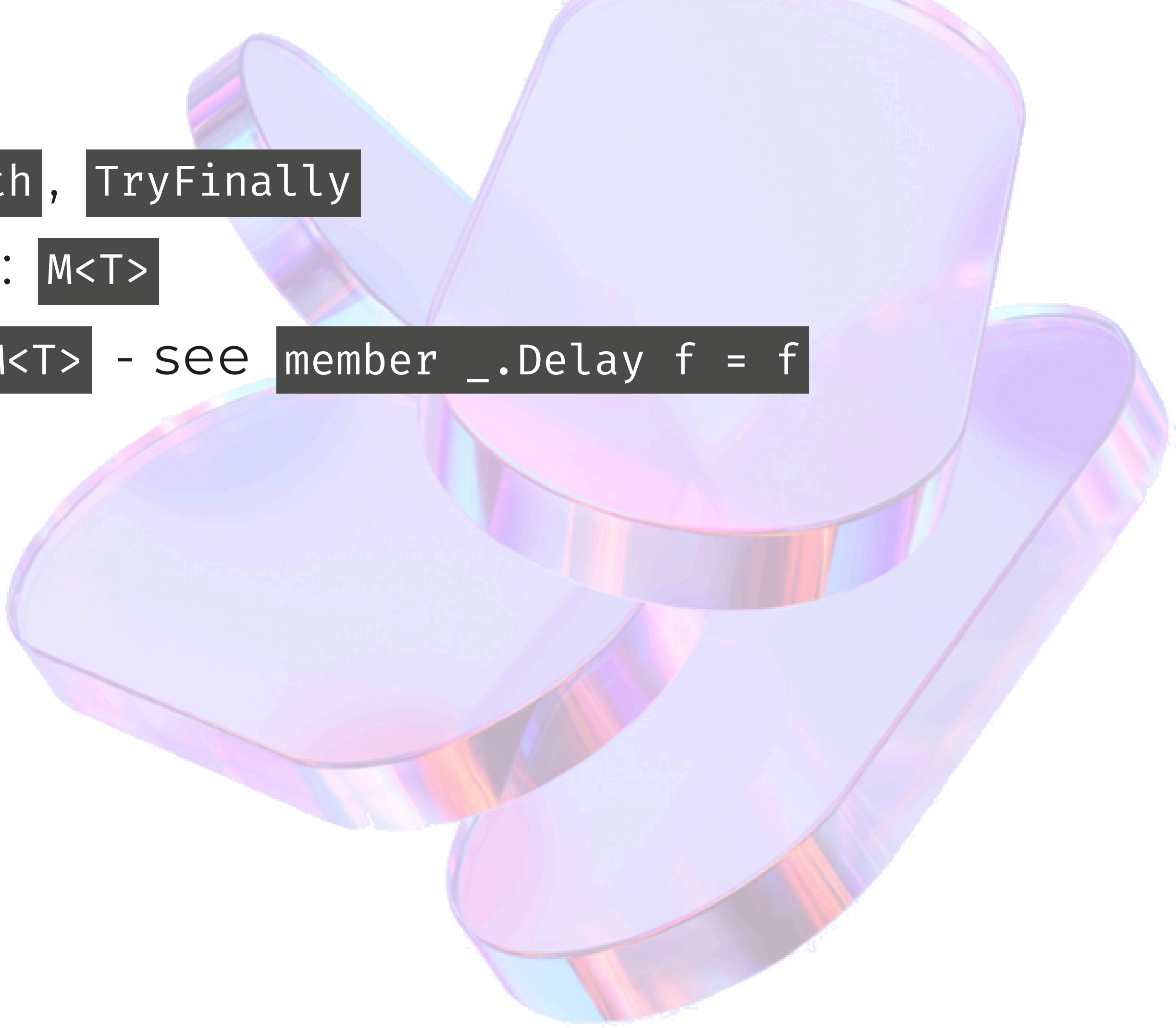
Examples of candidate types:

- Any generic type
- Any monoidal, monadic, or applicative type
- `string` as it contains `char`s
- Any type itself as `type Identity<'t> = 't` – see previous `logger {}` CE



`Delayed<T>` type

- Return type of `Delay`
- Parameter to `Run`, `Combine`, `While`, `TryWith`, `TryFinally`
- Default type when `Delay` is not defined: `M<T>`
- Common type for a real delay: `unit → M<T>` - see `member _.Delay f = f`



`Delayed<T>` type example: `eventually {}`

Union type used for both wrapper and delayed types:

```fsharp
// Code adapted from https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions
type Eventually<'t> =
    | Done of 't
    | NotYetDone of (unit → Eventually<'t>)

type EventuallyBuilder() =
    member _.Return x = Done x
    member _.ReturnFrom expr = expr
    member _.Zero() = Done()
    member _.Delay f = NotYetDone f

    member m.Bind(expr, f) =
        match expr with
        | Done x → f x
        | NotYetDone work → NotYetDone(fun () → m.Bind(work (), f))

    member m.Combine(command, expr) = m.Bind(command, (fun () → expr))

let eventually = EventuallyBuilder()
```

`Delayed<T>` type example: `eventually {}` (2)

The output values are maint to be evaluated interactively, step by step:

```fsharp
let step = function
    | Done x -> Done x
    | NotYetDone func -> func ()

let delayPrintMessage i =
    NotYetDone(fun () -> printfn "Message %d" i; Done ())

let test = eventually {
    do! delayPrintMessage 1
    do! delayPrintMessage 2
    return 3 + 4
}

let step1 = test |> step   // val step1: Eventually<int> = NotYetDone <fun:Bind@14-1>
let step2 = step1 |> step  // Message 1 ↩ val step2: Eventually<int> = NotYetDone <fun:Bind@14-1>
let step3 = step2 |> step  // Message 2 ↩ val step3: Eventually<int> = Done 7
```

`Internal<T>` type

`Return`, `ReturnFrom`, `Yield`, `YieldFrom`, `Zero` can return a type internal to the CE. `Combine`, `Delay`, and `Run` handle this type.

```fsharp
// Example: list builder using sequences internally, like the list comprehension does.
type ListSeqBuilder() =
    member inline _.Zero() = Seq.empty
    member inline _.Yield(x) = Seq.singleton x
    member inline _.YieldFrom(xs) = Seq.ofList xs
    member inline _.Delay([<InlineIfLambda>] thunk) = Seq.delay thunk
    member inline _.Combine(xs, ys) = Seq.append xs ys
    member inline _.For(xs, [<InlineIfLambda>] f) = xs ▷ Seq.collect f
    member inline _.Run(xs) = xs ▷ Seq.toList

let listSeq = ListSeqBuilder()
```

💡 Highlights the usefulness of `ReturnFrom`, `YieldFrom`, implemented as an *identity* function until now.

Builder methods without type (1)

— *Another trick regarding types* —
Any type can be turned into a CE by adding builder methods as extensions.

Example: `activity {}` CE to configure an `Activity` without passing the instance

- Type with builder extension methods: `System.Diagnostics.Activity`
- Return type: `unit` (no value returned)
- Internal type involved: `type ActivityAction = delegate of Activity → unit`
- CE behaviour:
  - monoidal internally: composition of `ActivityAction`
  - like a `State` monad externally, with only the setter(s) part

Builder methods without type (2)

```fsharp
type ActivityAction = delegate of Activity → unit

// Helpers
let inline private action ([<InlineIfLambda>] f: Activity → _) =
    ActivityAction(fun ac → f ac ▷ ignore)

let inline addLink link = action _.AddLink(link)
let inline setTag name value = action _.SetTag(name, value)
let inline setStartTime time = action _.SetStartTime(time)

type ActivityExtensions =
    [<Extension; EditorBrowsable(EditorBrowsableState.Never)>]
    static member inline Zero(_: Activity | null) = ActivityAction(fun _ → ())

    [<Extension; EditorBrowsable(EditorBrowsableState.Never)>]
    static member inline Yield(_: Activity | null, [<InlineIfLambda>] a: ActivityAction) = a

    [<Extension; EditorBrowsable(EditorBrowsableState.Never)>]
    static member inline Combine(_: Activity | null, [<InlineIfLambda>] a1: ActivityAction, [<InlineIfLambda>] a2: ActivityAction) =
        ActivityAction(fun ac → a1.Invoke(ac); a2.Invoke(ac))

    [<Extension; EditorBrowsable(EditorBrowsableState.Never)>]
    static member inline Delay(_: Activity | null, [<InlineIfLambda>] f: unit → ActivityAction) = f() // ActivityAction is already delayed

    [<Extension; EditorBrowsable(EditorBrowsableState.Never)>]
    static member inline Run(ac: Activity | null, [<InlineIfLambda>] f: ActivityAction) =
        match ac with
        | null → ()
        | ac → f.Invoke(ac)
```

Builder methods without type (3)

```
let activity = new Activity("Tests")

activity {
    setStartTime DateTime.UtcNow
    setTag "count" 2
}
```

- The `activity` instance supports the CE syntax thanks to its extensions.
- The extension methods are marked as not `EditorBrowsable` for proper DevExp.
- Externally, the `activity` is implicit in the CE body, like a `State` monad.
- Internally, the state is handled as a composition of `ActivityAction`.
- The final `Run` enables us to evaluate the built `ActivityAction`, resulting in the change (mutation) of the `activity` (the side effect).

CE creation guidelines 📜

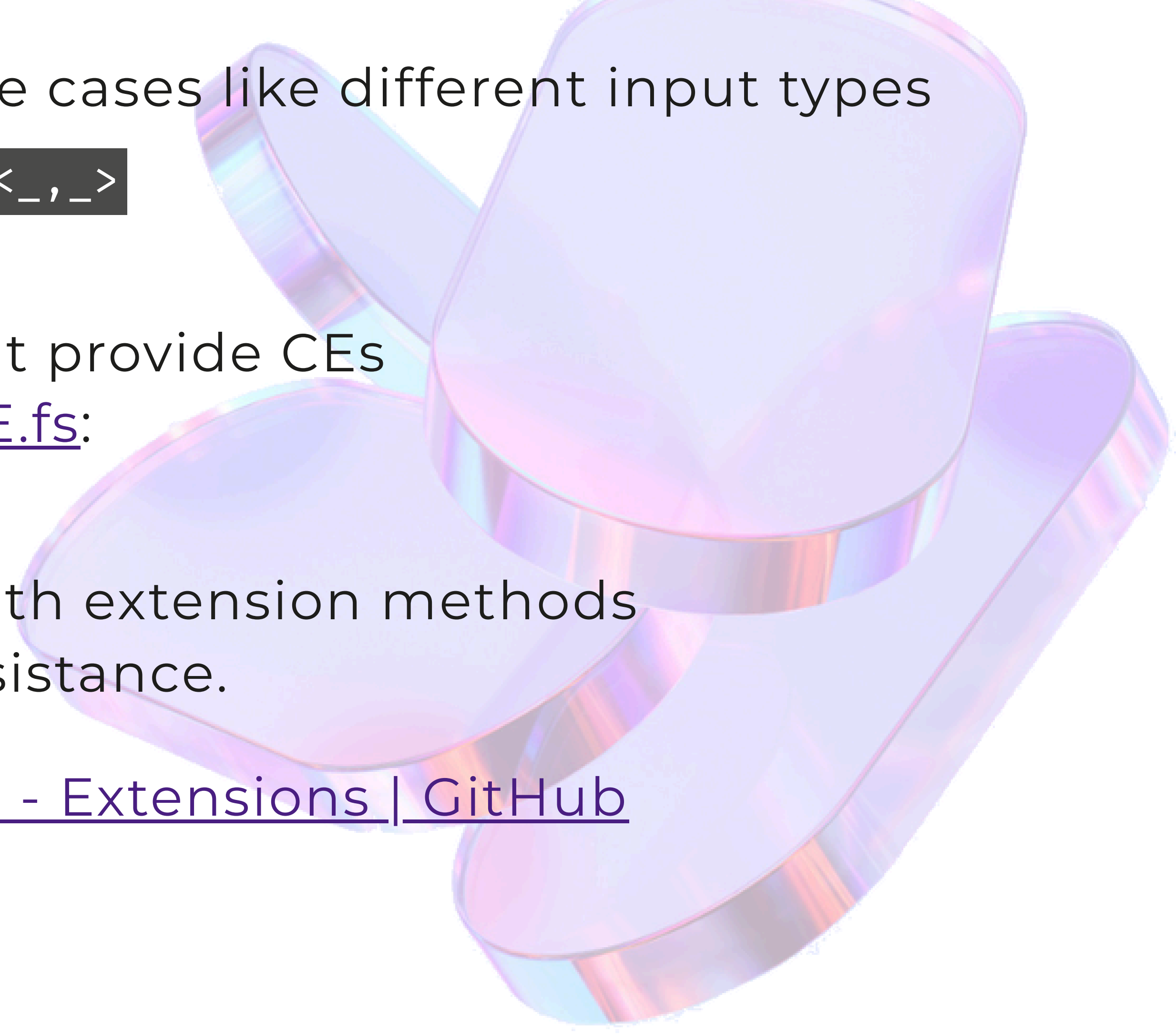
- Choose the main **behaviour**: monoidal? monadic? applicative?
  - Prefer a single behaviour unless it's a generic/multi-purpose CE
- Create a **builder** class
- Implement the main **methods** to get the selected behaviour
- Use/Test your CE to verify it compiles *(see typical compilation errors below)*, produces the expected result, and performs well.

```
1. This control construct may only be used if the computation expression builder defines a 'Delay' method
   ⇒ Just implement the missing method in the builder.
2. Type constraint mismatch. The type ''b seq' is not compatible with type ''a list'
   ⇒ Inspect the builder methods and track an inconsistency.
```

CE creation tips 💡

- Overload methods to support more use cases like different input types
  - `Async<Return<_,_>>` + `Async<_>` + `Result<_,_>`
  - `Option<_>` and `Nullable<_>`
- Get inspired by existing codebases that provide CEs
  → e.g. Tips found in FsToolkit/OptionCE.fs:
  - Undocumented `Source` methods
  - Force the method overload order with extension methods
    → to get better code completion assistance.

🔗 Computation Expressions Workshop: 6 - Extensions | GitHub



Custom operations 🚀

What: builder methods annotated with `[<CustomOperation("myOperation")>]`

Use cases: add new keywords, build a custom DSL
→ Example: the `query` core CE supports `where` and `select` keywords like LINQ

⚠️ **Warning:** you may need additional things that are not well documented:

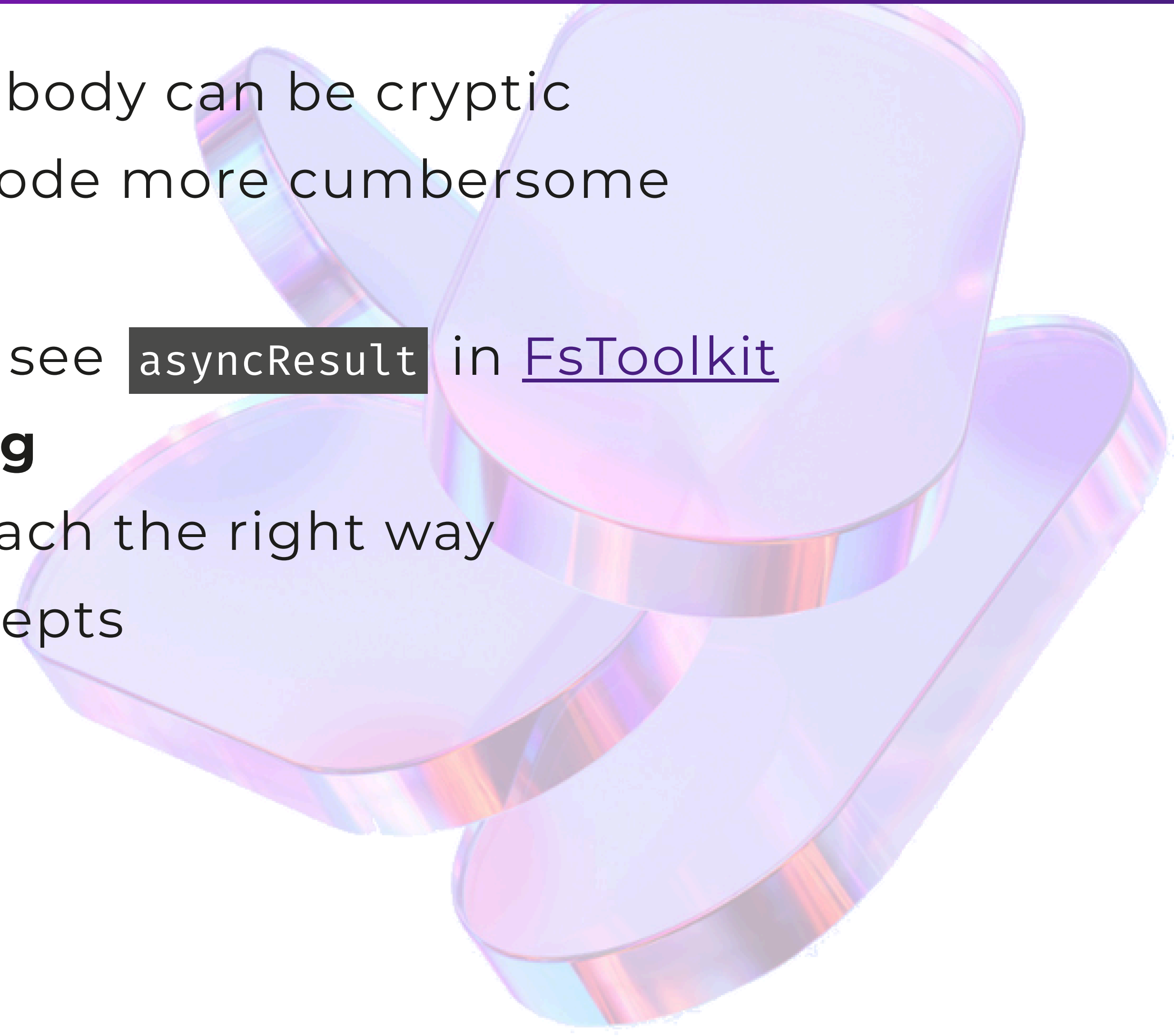
- Additional properties for the `CustomOperation` attribute:
  - `AllowIntoPattern`, `MaintainsVariableSpace`
  - `IsLikeJoin`, `IsLikeGroupJoin`, `JoinConditionWord`
  - `IsLikeZip`...
- Additional attributes on the method parameters, like `[<ProjectionParameter>]`

🔗 Computation Expressions Workshop: 7 - Query Expressions | GitHub

CE benefits ✅

- **Increased Readability**: imperative-like code
- **Reduced Boilerplate**: hides a "machinery"
- **Extensibility**: we can write our own "builder" for specific logic

CE limits ⚠️



- **Compiler error messages** within a CE body can be cryptic
- **Nesting different CEs** can make the code more cumbersome
  - E.g. `async` + `result`
  - Alternative: custom combining CE - see `asyncResult` in FsToolkit
- Writing our own CE can be **challenging**
  - Implementing the right methods, each the right way
  - Understanding the underlying concepts

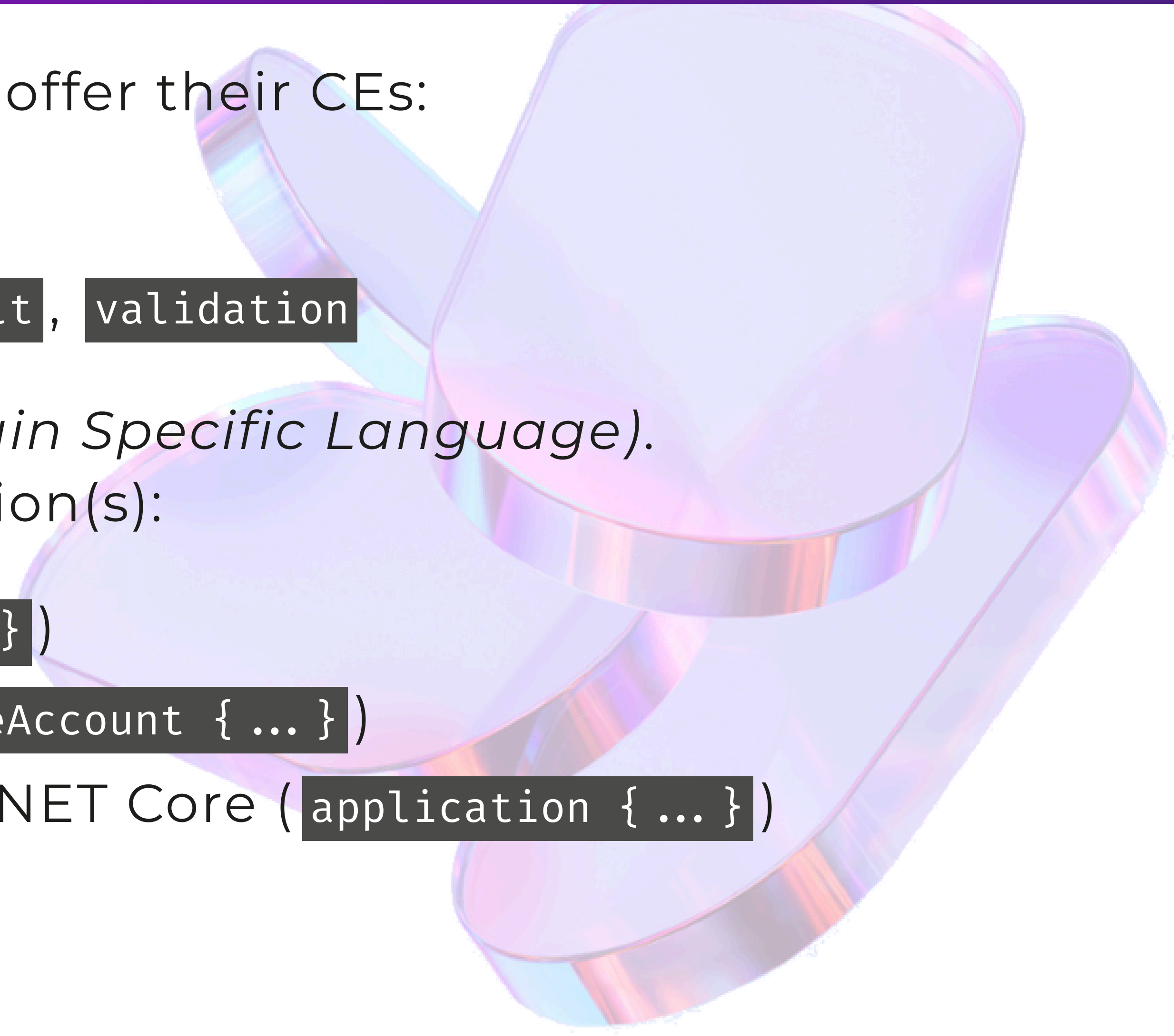
Other CEs

We've seen 2 libraries that extend F♯ and offer their CEs:

- FSharpPlus → `monad`
- FsToolkit.ErrorHandling → `option`, `result`, `validation`

Many libraries have their own DSL *(Domain Specific Language)*.
Some are based on computation expression(s):

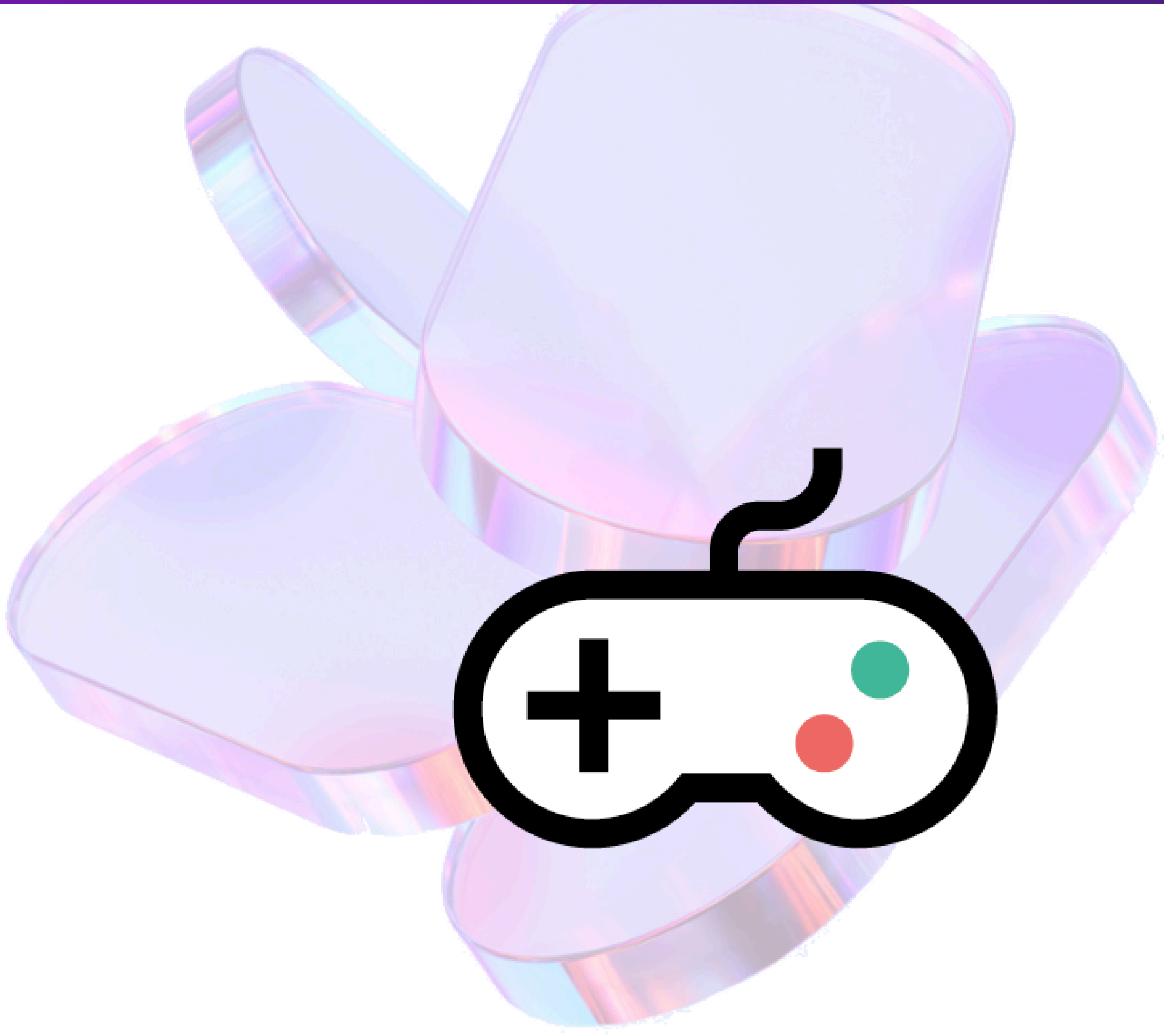
- Expecto: Testing library (`test "..." {...}`)
- Farmer: Infra as code for Azure (`storageAccount {...}`)
- Saturn: Web framework on top of ASP.NET Core (`application {...}`)



7. 🍔 Quiz

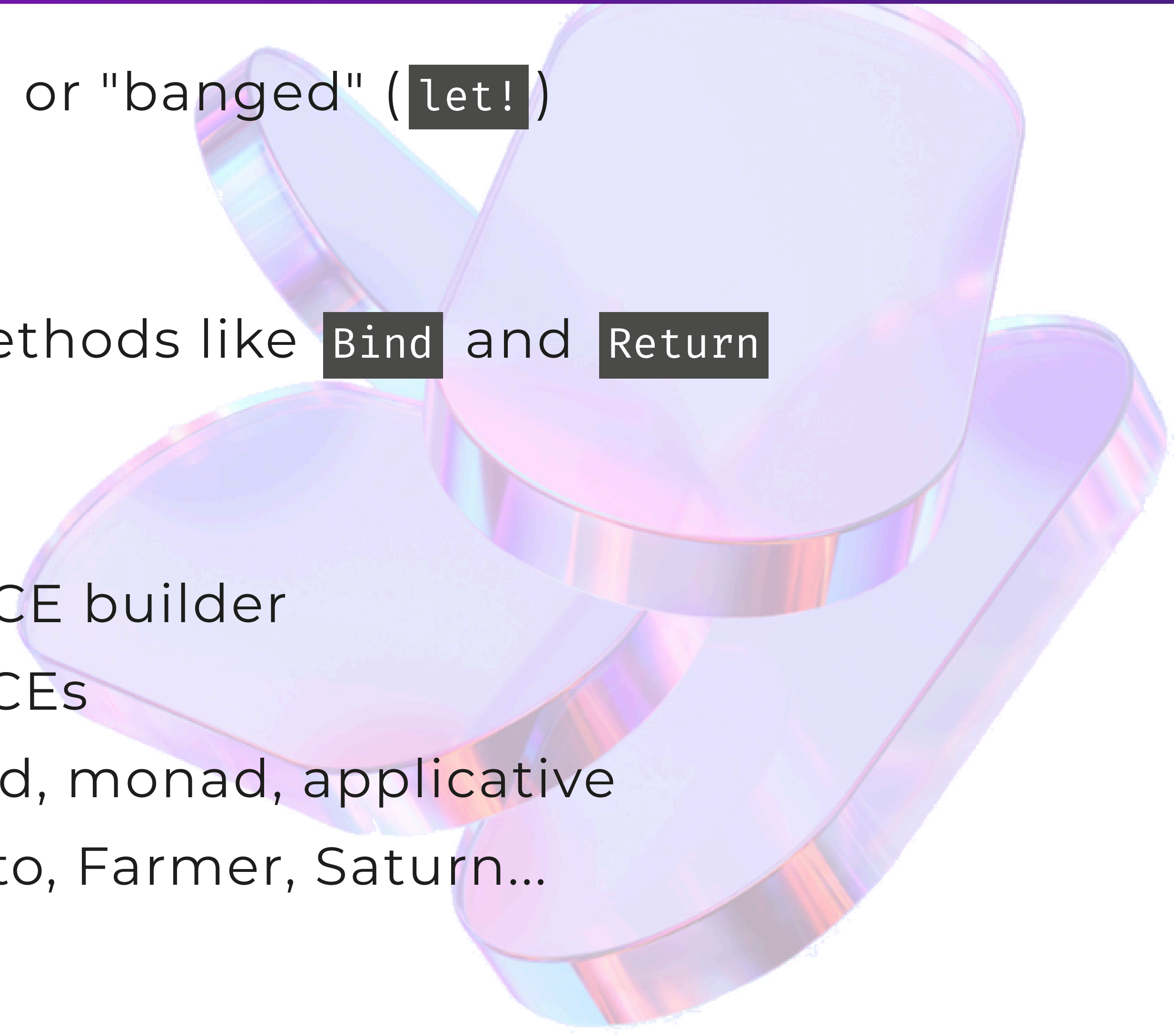
Quiz: Presentation

🔗 AhaSlides Quiz



8. Wrap up

Computation expression (CE)

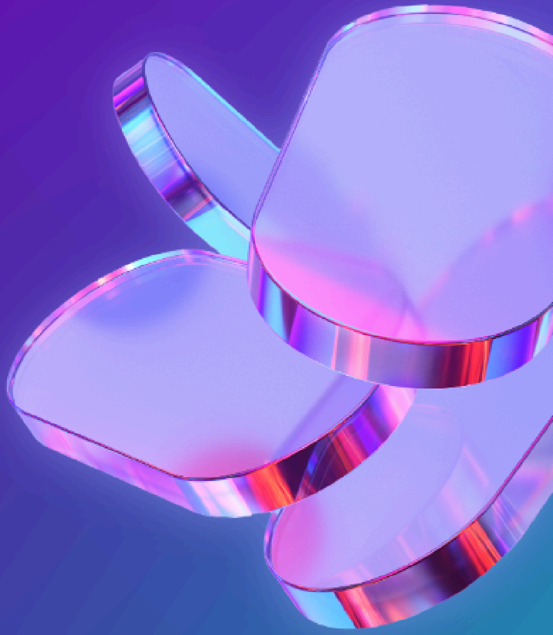


- Syntactic sugar: inner syntax: standard or "banged" (`let!`)
  → Imperative-like · Easy to use
- CE is based on a *builder*
  - instance of a class with standard methods like `Bind` and `Return`
- *Separation of Concerns*
  - Business logic in the CE body
  - Machinery behind the scene in the CE builder
- Little issues for nesting or combining CEs
- Underlying functional patterns: monoid, monad, applicative
- Libraries: FSharpPlus, FsToolkit, Expecto, Farmer, Saturn…

🔗 Additional resources

- Code examples in FSharpTraining.sln —Romain Deneau
- *The "Computation Expressions" series* —F# for Fun and Profit
- All CE methods | Learn F# —Microsoft
- Computation Expressions Workshop
- The F# Computation Expression Zoo —Tomas Petricek and Don Syme
  - Documentation | Try Joinads —Tomas Petricek
- Extending F# through Computation Expressions: 📹 Video • 📜 Article

Thanks 🙏



d-edge