Por qué molan los lenguajes que molan

@jjmerelo


Unicode
es
importante

Me suena a griego

```
function isελληνικά(input) {
  const ελληνικά = /^\p{Script=Greek}$/u
  return ελληνικά.test(input)
}
console.log(isελληνικά('π'));
```

λFTW

λ es lambda

```
from functools import reduce;

def fib(n):
    return reduce( lambda prev,this: prev+[ prev[-2]+prev[-1] ], \
                   range(1,n+1), \
                   [1,1] )

print(fib(12))
```

Inmutabilidad para programación rápida y segura

Lo que es inmutable no se puede cambiar

Escalando la inmutabilidad

```
val no = "me cambies"
```

¿Bucles? No, gracias.

```
(defn hasta-cero [n]
  (if (> n 0)
    (do
      (print n, "... ")
      (hasta-cero (dec n))
      )
    )
  )

(hasta-cero 3)
```

Tipado gradual: lo mejor de los dos mundos

Estático: `let secondNeedle: number = 3`

Dinámico: `let firstNeedle = "Hey"`

Juntos en TypeScript

```
let find = function ( needle, haystack ) {
    for (let i of haystack) {
        if ( needle === i ) {
            return true
        }
    }
    return false;
}
let randomList = ["Hey", 3]
let firstNeedle = "Hey"
let secondNeedle: number = 3
console.log( find( firstNeedle, randomList ) )
console.log( find( secondNeedle, randomList  ) )
```

Asignación postmoderna: desestructurando argumentos

Facilitando asignación en una sola orden

Y con seguridad

Desestructurando en Kotlin

```kotlin
data class Resultado (val resultado: Int, val tesoro: Boolean)
fun main(args: Array<String>) {
        val busca =  fun(intento: Int): Resultado
        {
                val cosas = listOf( 3, 33, 333, 42, 1, 1, 111 )
                if ( intento == 4  ) {
                        return Resultado( 42, true )
                } else {
                        return Resultado( cosas[intento], false )
                }
        }
        val (valor1, premio1) = busca( 2 )
        println( "2 devuelve " + valor1 + " y tiene premio " + premio1  )
        val (valor2, premio2) = busca( 4 )
        println( "4 devuelve " + valor2 + " y tiene premio " + premio2  )
}
```

Comparación de patrones para decisiones complejas

Evitando cascadas de `if`

Scalando la comparación

```
val puntos = (carta: String) => {
  carta match {
    case "As" => 11
    case "3"  => 10
    case "J"  => 2
    case "Q"  => 3
    case "K"  => 4
    case  _   => 0
  }
}

println("As ", puntos("A"))
println("7 ", puntos("7"))
```

Cuando hay que enganchar operaciones: cascadas

Tengo un Elixir para eso

```
resultado = 1..100
      |> Enum.filter(fn a -> rem(a,2) == 0 end)
      |> Enum.map(fn a -> :math.pow(a,2) end)
      |> Enum.reduce( &(&1 + &2) )

IO.puts  resultado
```

Envío múltiple ⇒ *Multiple dispatch*

Implementaciones rápidas a su servicio

Pintan bastos con Julia

```
@enum Suit ♣ ♦ ♥ ♠
@enum Palo Bastos Espadas Oros Copas

cadena( n::Int, s::Suit ) = string( n, " ", s )
cadena( n::Int, p::Palo ) = string( n, " de ", p )

println( cadena( 3, ♦) )
println( cadena( 7, Bastos ) )
```

Los perezosos heredarán la Tierra

Evaluación perezosa FTW

Pereza en F#

```fsharp
let horadam =
    (0.0f, 1.0f)
    |> Seq.unfold (fun (x, y) -> let z = 0.25f*x + 0.75f*y in Some(z, (y, z)))
    |> Seq.append [0.0f; 1.0f]


let seq_15 = horadam |> Seq.take 15
printfn "Primeros 15 %A" seq_15
```

Composición mejor que herencia

Traits, *mixins* y roles

Traits seguros con Rust

```
struct Carta { figura: String, palo: &'static str }

trait Mira {
    fn mira(self ) -> String;
}

impl Mira for Carta  {
    fn mira(self ) -> String {
        self.figura + " de " + self.palo
    }
}

fn main() {
    let as_de_bastos = Carta { figura: "As".to_string(),
                               palo: "Bastos"
    };
    println!( "Carta → {}", as_de_bastos.mira() );
}
```

JJ Merelo

@jjmerelo github.com/JJ

Pregunta inocente

Cuantos de estos conceptos se enseñan en las universidades

(Vale, quizás no tan inocente)

¿1ª vez que se menciona Perl 6?

No

Todas esas son funcionalidades de Perl 6

Tardé un rato en hacerlos

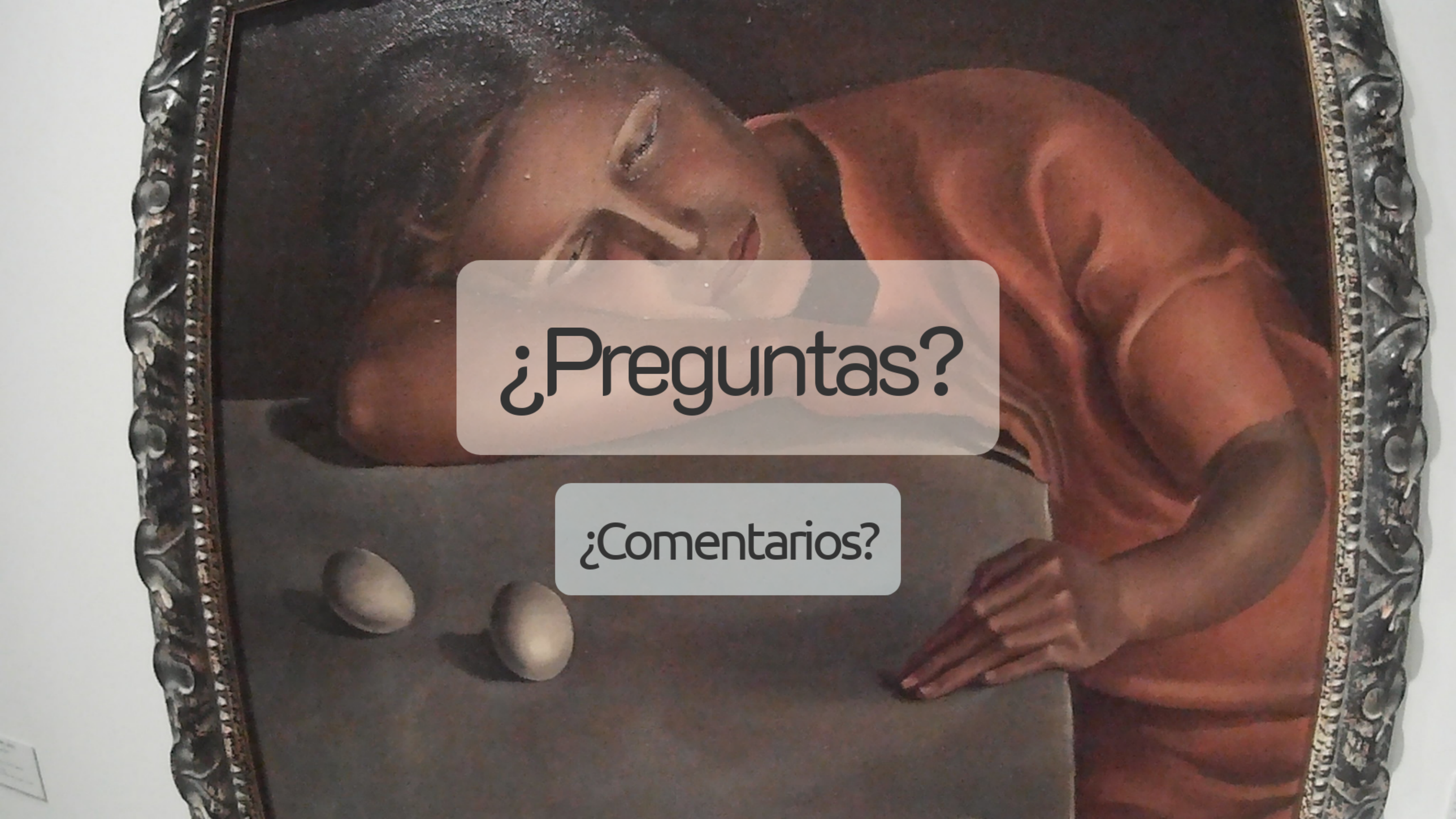
Pero no por el concepto, sino por la sintaxis

Perl 6: todo y además, el fregadero

```
multi sub collatz( 1 ) { return [1] }
multi sub collatz( Int $a where $a %% 2 ) { return  collatz( ($a/2).Int ).prepend( $a )}
multi sub collatz( $a where not $a %% 2 ) { return collatz( $a*3 + 1 ).prepend($a)}

my @collatz = lazy gather for \·····\ { take collatz( $_ ); }

1..100 ==> map( { @collatz[ $_ ] }) ==> grep( *.elems > 15 ) ==> my @long-chains;

sub prefix:<⇈> ( $𝑖 ) {
    given @collatz[ $𝑖 ].elems {
        when $_ > 15 { return @collatz[ $𝑖 ] but "Collatz", @collatz[ $𝑖 ].elems }
        default  { @collatz[ $𝑖 ], @collatz[ $𝑖 ].elems }
    }
}

for ^10 -> $þ {
    my ($seq, $elems) = ⇈$þ;
    say "Secuencia con $þ ", $seq.?Str eq "Collatz" ?? " es " !! " no es " , "Collatz";
}
```

Aprende Perl 6 para aprender *todos* los lenguajes

O al menos los que molan


¿Preguntas?
¿Comentarios?