Intro to Aiken

Aiken: Propiedades clave del lenguaje

- DSL para validadores en Cardano (Nodos de Cardano son el único target)
- Programación declarativa funcional (🚫 POO)
- Funciones de primera clase (todo es una expresión)
- Inmutabilidad
- Tipado estático con inferencia y genéricos
- Tipo de datos algebraico (ADTs)
- Recursión
- Puro
- Sistema de módulos

+ TIPOS PRIMITIVOS _

Aiken: Tipos primitivos - `Void`

`Void` sólo puede tener un valor: `Void`. Y, por eso, es un tipo que representa la falta de valor útil.

Análogos :

- En **Haskell** y **Rust** usamos `()` (unit) para el tipo y `()` para el valor.
- En **Java** usamos `Void` para el tipo y `null` para el valor.
- En **Swift** usamos `Void` para el tipo y `()` para el valor.

⚠️ Es un tipo útil en situaciones muy específicas (e.g., cuando querés indicar que no usas Redeemer), por lo que raramente se usa.

Aiken: Tipos primitivos - `Bool`

Un `Bool` es un valor que solo puede ser `True` o `False`

OPERADORES		
Operador	Descripción	Precedencia
`==`	Igualdad	4
`&&`	Conjunción lógica (“AND”)[1]	3
`||`	Disyunción lógica (“OR”)[1]	2
`!`	Nagación lógica (“NOT”)	1
`?`	Trazar si es falso	1

1* - asocian hacia la derecha con cortocircuito

También tenemos `and` y `or` para combinar cadenas de expresiones:

```
True && False && True || False
```

Es equivalente a

```
and {
  True,
  False,
  or { True, False}
}
```

Aiken: Tipos primitivos - `Int` - Sintaxis

`Int` es un entero de tamaño arbitrario (no underflow u overflow) y el único para valores numéricos.

Sintaxis:

Números literales	`42`
Números literales con separadores	`1_000_000 == 1000000`
Binario	`0b00001111 == 15`
Octal	`0o17 == 15`
Hexadecimal	`0xF == 15`

Aiken: Tipos primitivos - `Int` - Operadores

OPERADORES ARITMÉTICOS		
Operador	Descripción	Precedencia
+	Suma aritmética	6
-	Diferencia aritmética	6
/	División de enteros	7
*	Multiplicación aritmética	7
%	Resto de división de enteros	7

OPERADORES DE DESIGUALDAD		
Operador	Descripción	Precedencia
==	Igual[1]	4
>	Mayor	4
<	Menor	4
>=	Mayor o igual	4
<=	Menor o Igual	4

1* - Cualquier tipo serializable se puede comparar con ==.

Aiken: Tipos primitivos - ByteArray

ByteArray es un vector (array) de bytes.

Sintaxis:

Vector de bytes →

```
#[10, 255] // Usando base 10
#[1, 256] // error: 256 > 1 byte
#[0xff, 0x42] // Usando hex
```

String de bytes →

```
"foo" == #[0x66, 0x6f, 0x6f] == #[102, 111, 111]
```

String de bytes en Hex →

```
#"666f6f" == #[0x66, 0x6f, 0x6f] == "foo"
```

Aiken: Tipos primitivos - `String`

El tipo String es texto encoded en binarios UTF-8 que pueden contener caracteres de Unicode.

Sintaxis:

En una línea → `@"こんにちは ALBA!🤩"`

Multilínea →

```
@"こんにちは
ALBA!🤩"
```

⚠️ Sólo usamos String para trazado (depurar). Cuando se te ocurra usar String para otra cosa, usa ByteArray.

Aiken: Tipos primitivos - `Data`

El tipo Data es un tipo **opaco** que representa cualquier valor serializable. Cualquier tipo definido por el usuario se puede convertir a Data y viceversa de forma segura.

Cuándo se usa?

- Cuando querés utilizar valores de diferentes tipos en una estructura homogénea.
- Cuando no querés restringir un valor a una estructura específica.
- Cuando usas funciones integradas que solo funcionan con Data como forma de lidiar con polimorfismo.

Aiken: Tipos primitivos - List<a>

Una lista (List<a>) es una colección ordenada de valores homogéneos. La variable de tipo a se utiliza para representar cualquier tipo posible.

Sintaxis:

```
[1, 2, 3, 4]     // List<Int>
["text", 3, 4]   // Error de tipeo!
```

Insertar al frente de la lista es la forma preferida de agregar nuevos valores:

```
let x = [2, 3]     // x == [2, 3]
let y = [1, ..x]   // y == [1, 2, 3]
```

Aiken: Tipos primitivos - Tuple

Las tuplas son útiles para agrupar valores de distintos tipos.

Sintaxis:

```
(10, "hello") // El tipo es (Int, ByteArray)
(1, 4, [0])   // El tipo es (Int, Int, List<Int>)
```

Se puede acceder a los elementos utilizando el punto seguido del índice del elemento (ordinal):

```
let valor = (14, 42, 1337)
let a = valor.1st
let b = valor.2nd
let c = valor.3rd
(c, b, a) // (1337, 42, 14)
```

⚠️ Por lo general, se desaconsejan las tuplas largas (es decir, más de 3 elementos).

Aiken: Tipos primitivos - Tipos avanzados

Tipos necesarios para comportamientos más avanzados. Vamos a usar algunos de estos más adelante.

Par (`Pair<a, b>`) → Similar a una tupla de 2 valores (`(a, b)`), pero se construye distinto (`Pair(14, "aiken")`) y su representación subyacente es distinta.

Pares (`Pairs<a, b> = List<Pair<a, b>>`) → Existe por conveniencia ya que las listas de pares son comunes.

`PRNG` & `Fuzzer` → Vamos a hablar de Pseudo-Random Number Generator y Fuzzer en la lección de testing.

`G1Element`, `G2Element` & `MillerLoopResult` → Específico para el uso de primitivas criptográficas de la curva BLS12-381.

+ VARIABLES Y CONSTANTES _

Aiken: Variables y Constantes - `let`

Aiken usa la keyword let para vincular nombres a valores:

```
let x = 1
let y = 2

y + x == 3
```

Los valores vinculados al nombre son **inmutables**; sin embargo, los vínculos nuevos pueden eclipsar los enlaces anteriores:

```
let x = 1
let y = x
let x = 2

y + x == 3
```

Aiken: Variables y Constantes - `expect`

`expect` es una keyword que funciona como `let`, pero permite realizar algunas conversiones potencialmente inseguras.

```
expect [valor] = lista_de_valores
```

`expect` es una keyword especial porque es una de las pocas que permite tener **efectos secundarios**. **Si el valor no encaja con la forma especificada, la transacción falla.**

Vamos a ver más ejemplos en la sección de tipos personalizados.

Aiken: Variables y Constantes - `const`

Aiken no permite vínculos `let` a nivel del módulo (Aiken tiene modulos). Para compartir valores fijos en múltiples lugares de nuestro proyecto, usamos `const`:

```
const year = 2024
const config_policy_id = #"00000000000000000000000000000000000000000000000000000000"
const token_name = @"ALBA"
```

Como **todos** los valores en Aiken, las constantes del módulo son **inmutables** y no se pueden utilizar como estado mutable global. Cuando se hace referencia a una constante, el compilador inserta el valor como si lo hubieras escrito ahí.

⚠️ Solo puede declarar constantes de módulo para los tipos: `Int`, `ByteArray` y `String`.

Aiken: Variables y Constantes - Anotar tipos

A las variables y constantes se les pueden dar anotaciones de tipo. Estas anotaciones sirven como documentación o pueden usarse para proporcionar un tipo más específico de lo que el compilador inferiría:

```
const name: ByteArray = "Aiken"
const size: Int = 100

let result: Bool = 14 > 42
```

+ FUNCIONES_

Aiken: Funciones - Funciones con nombre

Las funciones se definen utilizando la keyword fn. Pueden tener o no argumentos, pero siempre devuelven un valor.

```
fn sumar_uno(x: Int) -> Int {
  x + 1
}

/// Esta función toma otra función como argumento
fn dos_veces(f: fn(t) -> t, x: t) -> t {
  f(f(x))
}

/// Esta función usa otra función del ámbito
fn sumar_dos(x: Int) -> Int {
  dos_veces(sumar_uno, x)
}
```

Es buena práctica escribir los tipos de argumentos y retorno. Proporciona documentación y ayuda a pensar en los tipos a medida que se escribe el código.

Código con tipos y sin tipos es igual de seguro.

Aiken: Funciones - Funciones anónimas

Las funciones anónimas tienen prácticamente la misma sintaxis:

```
fn run() {
  let sumar = fn(x, y) { x + y }
  sumar(1, 2)
}
```

Como en todos los lenguajes, las funciones anónimas son especialmente útiles cuando solo se usan una vez.

Aiken: Funciones - Etiquetado de argumentos

Los argumentos de las funciones se pueden pasar posicionalmente o usando etiquetas:

```
// Función que reemplaza un patrón en un String
fn replace(self: String, pattern: String, replacement: String) -> String {
  // ...
}

// Usando argumentos posicionales
replace(@"A,B,C", @",", @" ")
// Usando argumentos con etiquetas
replace(self: @"A,B,C", pattern: @",", replacement: @" ")
// Los argumentos con etiquetas pueden pasarse en cualquier orden
replace(pattern: @",", replacement: @" ", self: @"A,B,C")
// Mezclando argumentos posicionales y con etiquetas
replace(@"A,B,C", pattern: @",", replacement: @" ")
```

Aiken: Funciones - Pipe operator

Aiken provee el operador |> de precedencia 0 para pasar el resultado de una función como argumento de otra. Similar al mismo operador en Elixir y F#.

```
to_string(reverse(from_string(string)))
```

```
string
    |> from_string
    |> reverse
    |> to_string
```

Aiken: Funciones - Captura de funciones

Existe una sintaxis abreviada para crear funciones anónimas que toman un argumento y llaman a otra función.

```
fn sumar(x, y) {
  x + y
}

fn run() {
  let sumar_uno = sumar(1, _)
  sumar_uno(2)
}
```

El _ se utiliza para indicar dónde se debe pasar el argumento.

Aiken: Funciones - Captura de funciones con |>

La sintaxis de captura de función se usa a menudo con pipe |> para crear una serie de transformaciones. De hecho, este uso es tan común que existe una abreviatura especial:

```
fn sumar(x: Int , y: Int ) -> Int {
  x + y
}
```

```
fn sumar() {
  // sumar(sumar(sumar(1, 3), 6), 9)
  1
  |> sumar(_, 3)
  |> sumar(_, 6)
  |> sumar(_, 9)
}
```



```
fn sumar() {
  1
  |> sumar(3)
  |> sumar(6)
  |> sumar(9)
}
```

El pipe operator primero verifica si el valor a su izquierda puede usarse como primer argumento de la llamada, e.g. a |> b(1, 2) se vuelve b(a, 1, 2). Si no, vuelve a llamar al resultado del lado derecho como una función, e.g. b(1, 2)(a).

+ TIPOS PERSONALIZADOS _

Aiken: Tipos Personalizados - Intro

Si venís de programación funcional:

Los tipos de Aiken son algebraicos (ADTs).

Si venís de programación orientada a objetos:

Lost tipos de Aiken son como objetos que no tienen métodos

Aiken: Tipos Personalizados - Sintaxis

Se definen con la keyword type:

```
type Datum {
  Datum { firma: ByteArray, cantidad: Int }
}
```

```
type Datum {
  signer: ByteArray,
  count: Int
}
```

Y se crean valores usando el constructor:

```
fn datums() {
  // Los campos pueden estar en cualquier orden
  let datum1 = Datum { firma: #[0xAA, 0xBB], cantidad: 2001 }
  let datum2 = Datum { cantidad: 1805, firma: #[0xAA, 0xCC] }

  [datum1, datum1]
}
```

Aiken: Tipos Personalizados - Ejemplos

Descripción	Tipo	Valor

Un constructor con 2 campos.	`type Datum {`
	`  signer: ByteArray,`
	`  count: Int`
	`}`
		`let valor = Datum {`
		`  signer: #[0xf6, 0xff],`
		`  count: 2001`
		`  }`

Dos constructores. Ambos con zero campos.	`type Bool {`
	`  True`
	`  False`
	`}`
		`let valor = True`

Dos constructores. El primero con un campo y el segundo sin campos.	`type User {`
	`  LoggedIn { count: Int }`
	`  Guest`
	`}`
		`let usuario1 = LoggedIn { count: 4 }`
		
		`let visitante = Guest`

Dos constructores. El primero sin campos y el segundo con un campo genérico.	`type Option<a> {`
	`  None`
	`  Some(a)`
	`}`
		`let no_hay_nada = None`
		
		`let hay_algo = Some(4)`

Aiken: Tipos Personalizados - Destructuring/Pattern Matching

Una vez tenemos un tipo personalizado, podemos hacer coincidir patrones (pattern match) para asignar nombres a los valores internos y elegir camino de ejecución:

```
fn get_name(user) {
  when user is {
    LoggedIn { count } -> count
    Guest -> "Guest user"
  }
}
```

```
type Score {
  Points(Int)
}

let score = Points(50)
// Esto trae a `p` al ambito.
let Points(p) = score

p // 50
```

Aiken: Tipos Personalizados - Destructuring - Ignorar Valores

Al desestructurar los campos de un constructor, uno puede ignorar valores específicos con _ o ignorar todo lo que no fue explícitamente especificado con .. (doble punto):

```
// Creo un tipo
type Perro {
  Perro { nombre: ByteArray, ternura: Int, edad: Int }
}

// Creo un valor
let perro = Perro { nombre: #"44616e61", ternura: 9001, edad: 6 }

//  Especifico todos los campos pero ignoro 2.
let Perro { nombre: nombre, ternura: _, edad: _ } = perro
builtin.decode_utf8(nombre) // "Dana"

// Especifico solo `edad` e ignoro el resto
let Perro { edad, .. } = perro  // Field punning: `edad` == `edad: edad`.
edad // 6
```

Aiken: Tipos Personalizados - Acceder a valores sin desestructurar

Si un tipo tiene solo una variante y campos con nombre, se puede acceder a ellos usando .nombre_campo:

```
type Perro {
  Perro { nombre: ByteArray, ternura: Int, edad: Int }
}

let perro = // ...obtengo un valor de tipo `Perro`.
perro.ternura // Esto devuelve algo seguro


type User {
  LoggedIn { count: Int }
  Guest
}

let usuario = // ..obtengo un valor de tipo `User`
usuario.count // ERROR de compilado: Como se que `count` existe?
```

Aiken: Tipos Personalizados - Actualización de registro (Record update)

Aiken proporciona una sintaxis que permite actualizar un subset de los campos de un record:

```
type Person {
  name: ByteArray,
  shoe_size: Int,
  age: Int,
  is_happy: Bool,
}

fn have_birthday(person: Person) -> Person {
  // Es el cumpleaños de la persona, así que aumentamos su edad en 1
  // y la hacemos feliz.
  Person { ..person, age: person.age + 1, is_happy: True }
}
```

Los valores son **inmutables**. Por lo que la sintaxis de actualización crea un **nuevo valor** con los campos del registro inicial pero reemplazando los especificados con sus nuevos valores.

Aiken: Tipos Personalizados - Alias

Un alias de tipo te permite crear un nombre alternativo para un tipo, sin ninguna información adicional. Útiles para simplificar tipos y hacer el código más legible:

```
type Edad = Int


type Persona = (String, Edad)


fn crear_persona(nombre: String, edad: Edad) -> Persona {
  (nombre, edad)
}
```

+ ESTRUCTURAS DE CONTROL_

Aiken: Estructuras de control - Bloques

Cada bloque de Aiken es una expresión. Se ejecutan todas las expresiones del bloque y se devuelve el resultado de la última expresión:

```
let value: Bool = {
    "Hello"
    42 + 12
    False
}

value == False
```

En este caso, como no hay efectos secundarios, las linas con ”Hello” y 42 + 12 no cambian nada.

Aiken: Estructuras de control - Pattern Matching

La forma más común de controlar el flujo del programa es haciendo pattern matching con la expression when *expr* is:

```
let resultado =
  when lista is {
    []       -> @"Lista vacia"
    [a]      -> @"Lista con 1 elemento"
    [a, 42] -> @"Lista con 2 elementos y el segundo es 42"
    [a, b]   -> @"Lista con 2 elementos"
    [a, b, 56, ..] -> @"Lista con 3 o más elementos donde el tercero es 56"
    _otro -> @"Lista con cualquier otra cantidad de elementos"
  }
```

Asignar nombres a subpatrones

```
when ys is {
    [[_, ..] as lista_interna] -> lista_interna
    _otro                      -> []
  }
```

Aiken: Estructuras de control - If-Else

También tenemos la expression if-else que puede ser encadenada las veces que sea necesario:

```
let un_booleano = True

let resultado =
  if un_booleano {
    "Es verdadero!"
  } else {
    "Es falso."
  }

resultado // "Es verdadero!"
```

```
fn fibonacci(n: Int) -> Int {
  if n == 0 {
    0
  } else if n == 1 {
    1
  } else {
    fibonacci(n - 2) + fibonacci(n - 1)
  }
}
```

Si bien puede parecer código imperativo, en realidad, if-else es una sola expresión. Esto significa que los **tipos de retorno de ambas ramas deben coincidir**.

+ Effectos 👀_

Aiken: Efectos - Intro

En los lenguajes de alto nivel que compilan a UPLC, teóricamente no necesitamos efectos. Sinembargo, usualmente, y en el caso de Aiken, tenemos dos:

- **Cancelar la transacción:** Muchas veces no es necesario evaluar la expresión entera para saber que va a fallar o llegamos a estados que consideramos inválidos. En estos casos, `fail` y `expect` son útiles.

- **Agregar un trace:** No es un efecto que cambie el resultado de la evaluación, sino uno que permite investigar las razones de porque falló (si esque falló). Para esto, usamos `trace` y `?`.

Aiken: Efectos - Trazado - `trace`

`trace` nos permite trazar mensajes para que la máquina virtual de Plutus capture en momentos específicos. Sirven únicamente para depurar los validadores.

```
fn es_par(n: Int) -> Bool {
  trace "es_par"
  n % 2 == 0
}

fn es_impar(n: Int) -> Bool {
  trace "es_impar"
  n % 2 != 0
}
```

```
let n = 10
es_par(n) || es_impar(n)
```

⚠️ Los trazados se remueven por defecto al compilar y se dejan por defecto al testear!

Aiken: Efectos - Trazado - ? (trazar-si-falso)

Como los validadores son fundamentalmente predicados, lo común es que se estructuren como conjunciones y disyunciones de expresiones booleanas. Por eso, Aiken provee ?:

```
fn checkear_todo() {
  let a_tiempo = checkear_fecha_limite()
  let gasto_el_token = checkear_token()
  let enviado_a_benef = checkear_beneficiario()

  a_tiempo && gasto_el_token && enviado_a_benef
}
```

→ `False`

```
fn checkear_todo() {
  let a_tiempo = checkear_fecha_limite()
  let gasto_el_token = checkear_token()
  let enviado_a_benef = checkear_beneficiario()

  a_tiempo? && gasto_el_token? && enviado_a_benef?
}
```

→ `"a_tiempo ? False"`

Aiken: Efectos - Cancelar transacción - `fail` y `todo`

Función que me devuelve un valor sin el cual no puedo evaluar la validez de la transacción:

```
fn extraer(opt: Option<a>) -> a {
  when opt is {
    Some(a) -> a
    None -> fail // Fallar ahora ya!
  }
}
```

```
fn extraer(opt: Option<a>) -> a {
  when opt is {
    Some(a) -> a
    None -> fail @"No tengo valor!"
  }
}
```

```
fn extraer(opt: Option<a>) -> a {
  when opt is {
    Some(a) -> todo @"implementar una vez que.."
    None    -> fail @"No tengo valor!"
  }
}
```

Aiken: Efectos - Cancelar transacción - `expect`

La todopoderosa keyword `expect` es una de las más útiles en Aiken. Con `expect` podemos "forzar" valores a tener una estructura completa. Y si no, falla la transacción como con `fail`.

Emparejamiento de patrones no exhaustivo

```
let valor_interno =
  when opt is {
    Some(a) -> a
    None -> fail
  }

expect Some(valor_interno) = opt

// ..uso valor_interno
```

Transformar `Data` a una representación concreta

```
type MyDatum {
  foo: Int,
  bar: ByteArray,
}

fn to_my_datum(data: Data) -> MyDatum {
  expect my_datum: MyDatum = data
  my_datum
}
```

+ Módulos_

Aiken: Módulos - Intro

- Los programas y librerías de Aiken se componen de conjuntos de funciones y tipos llamados **módulos**.

- Cada **módulo** tiene su propio espacio de nombres y puede exportar **tipos** y **valores** para ser utilizados por otros módulos del programa.

Aiken: Módulos - Crear módulo

Crear archivo de aiken con el código que queres que sea parte del módulo. Usualmente los modulos estan dentro de una carpeta con el mismo nombre que el proyecto:

```
// Crear archivo: lib/comida_domingo/mate.ak
// Dentro de ese archivo:

fn calentar_agua(agua_fria, pava) {
  todo
}

fn poner_yerba(mate) {
  todo
}

pub fn preparar_mate(agua_fria, pava, mate) {
  let termo = calentar_agua(agua_fria, pava)
  let mate_listo = poner_yerba(mate)
  (termo, mate_listo)
}
```

Aiken: Módulos - Importar módulo

Usamos la keyword use seguida del camino al módulo que queremos importar:

```
// Dentro del módulo lib/comida_domingo/tardecita.ak
use comida_domingo/mate
pub fn merienda(cocina) {
  // ...
  let (termo, mate_listo) = mate.preparar_mate(agua_fria, pava, mate)
  // ...
}
```

Aiken: Módulos - Importar módulo - Cambiar nombre de espacio y sin espacio

```
use comida/carne
use comida_domingo/carne as asado
use morfi/vegetariano.{ Ensalada, vegetales, cortar, condimentar}

pub fn comida_vegana() -> Ensalada {
   vegetales
    |> cortar()
    |> condimentar()
}

pub fn comida_carnivoros() -> carne.Carne {
    asado.asar(carne.carne)
}
```

Aiken: Módulos - Tipos opacos

```
// Constructor no disponible fuera del módulo
pub opaque type Contador {
  Contador(valor: Int)
}

pub fn new() {
  Contador(0)
}

pub fn incrementar(contador: Contador) {
  Contador(contador.valor + 1)
}
```

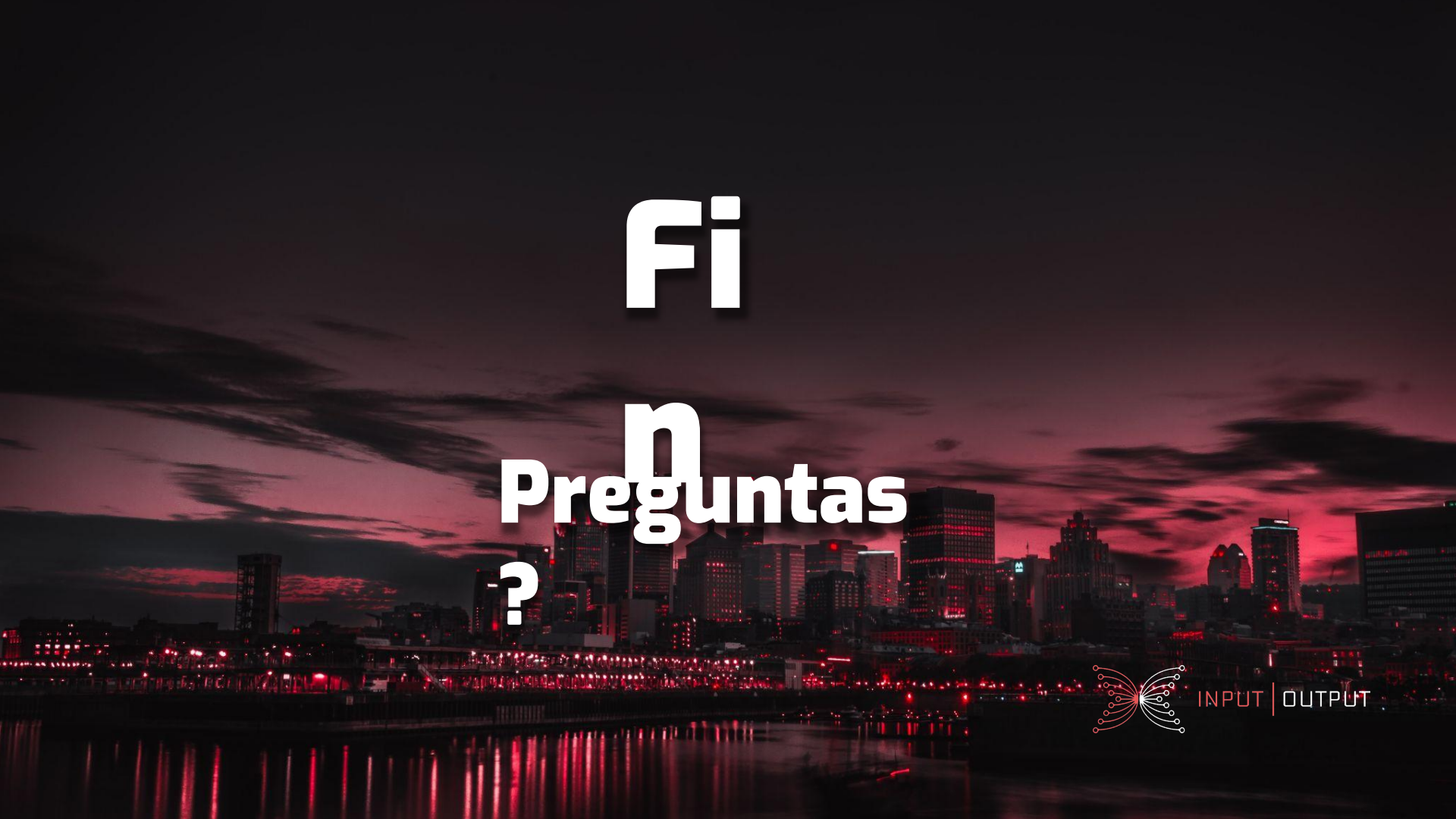
Aiken: Módulos - Prelude, Built-ins y Stdlib

Hay ciertos módulos que ya vienen o como parte del lenguaje o son facilitados por la CLI que crea un proyecto nuevo:

- **Prelude:** Hay dos módulos integrados en el lenguaje, el primero es `prelude`. Este provee código básico (como la definición de `Bool`). Se importa automáticamente, pero se puede optar por importarlo de la forma habitual en caso de colisión de nombres.

- **Built-ins:** Segundo módulo que viene con el lenguaje. Expone funciones integradas de Plutus Core que son útiles. Lo mas seguro es que casi nunca uses ésta librería directamente, sino a traves de la libreria estandar.

- **Stdlib:** Librería estandar. Compuesta de un conjunto de módulos que son útiles para cualquier proyecto. Dónde está la mayoría de funciones que vas a usar.


Fin
Preguntas
?
INPUT | OUTPUT