¿Qué es una sealed class en Kotlin?

Una sealed class permite declarar una jerarquía cerrada de subtipos. Eso significa que todas las subclases de Valor están definidas en el mismo archivo, lo que permite al compilador verificar que todas las posibilidades están cubiertas (por ejemplo, en un when).

¿Qué es una data class en Kotlin?

Es una clase diseñada específicamente para mantener y representar datos. Kotlin proporciona automáticamente una serie de funcionalidades útiles cuando se declara una clase como data, lo que reduce el código repetitivo que normalmente se escribiría en clases que simplemente almacenan información.

¿Qué es una object en Kotlin?

Se utiliza para declarar una instancia única (singleton) de una clase, o para definir objetos anónimos (similar a clases anónimas en Java). Se puede usar en diferentes contextos, pero su objetivo principal es definir una única instancia accesible globalmente, sin necesidad de crearla explícitamente.

1. Representación de las expresiones: Expresion

```
sealed class Expresion {
   data class Atomo(val valor: String) : Expresion()
   data class Lista(val elementos: List<Expresion>) : Expresion()
}
```

¿Por qué usar una sealed class?

-Permite representar una jerarquía cerrada de tipos.

-Todas las subclases están definidas en el mismo archivo, lo cual permite que los when sobre Expr sean exhaustivos.

-Es útil para representar árboles de sintaxis como los S-expressions de Racket.

¿Por qué usar data class?

-Proporciona automáticamente equals, hashCode, y toString, muy útiles para depurar y comparar nodos del árbol.

-Atomo: un identificador, número o palabra.

-ListExpr representa listas de expresiones.

2. Parser:

ParserSExpresion se encarga de convertir un código fuente escrito en notación S (S-expressions), como, por ejemplo:

(define x 42) en una estructura de datos jerárquica (objetos Expresion) que puede ser procesada por el evaluador (Evaluador).

Implementa la interfaz IParserExpr, por lo tanto, garantiza que expone la función parsear.

parsear(entrada: String): List<Expresion>

Función pública.

Responsable de iniciar el proceso de parsing. Hace lo siguiente:

Llama a tokenizar() para dividir el texto en tokens.

Usa un bucle while para ir leyendo expresiones completas con leerDesdeTokens().

Devuelve una lista de expresiones Expresion, una por cada S-expression de la entrada.

Ejemplo:

val entrada = "(define x 42) (define y 99)"

val resultado = parser.parsear(entrada)

resultado será una lista con dos expresiones: ***Lista(Atomo("define"), Atomo("x"), Atomo("42")).***

tokenizar(str: String): List<String>

Función privada.

Convierte el string en una lista de tokens léxicos.

Cómo funciona:

Separa los paréntesis añadiendo espacios alrededor de (y).

Divide por espacios usando una expresión regular \\s+.

Filtra los tokens vacíos con filter { it.isNotEmpty() }.

leerDesdeTokens(tokens: MutableList<String>): Expresion

Función recursiva privada.

Toma tokens y los convierte en objetos Expresion recursivamente.

Lógica:

Si el token es "(", crea una nueva lista de expresiones:

Lee tokens uno por uno hasta encontrar ")".

Cada token intermedio se convierte recursivamente en una Expresion (puede ser lista o átomo).

Si el token es ")" sin par anterior, lanza error.

En cualquier otro caso, devuelve un Atomo.

Ejemplo de parsing:

Entrada: ["(", "define", "x", "42", ")"]

Salida: Lista([Atomo("define"), Atomo("x"), Atomo("42")])

Estructura de datos:

Usa MutableList<String> como buffer destructivo (consume tokens).

Construye árboles con Expresion.Lista y Expresion.Atomo.

3. Clase Valor

La clase Valor es una jerarquía sellada (sealed class) que representa todos los tipos de valores que puede manejar el intérprete durante la ejecución de programas en el lenguaje.

Esta estructura es fundamental porque define cómo el evaluador y el entorno entienden y manipulan resultados en tiempo de ejecución.

¿Qué representa Valor?

Valor es una representación en tiempo de ejecución de los distintos tipos de datos que puede manejar el lenguaje interpretado: números, booleanos, símbolos, funciones (nativas o definidas por el usuario), y un valor especial Indefinido.

Numero(Int), Booleano(Boolean), Simbolo(String)

Funcion: Funciones definidas por el usuario (lambda).

FuncionNativa: Funciones nativas implementadas en Kotlin.

Indefinido: Representa un valor sin definir (como en define).

4. abstract class FuncionIncorporada

¿Qué es FuncionIncorporada?

Esta clase:

Hereda de Valor, por lo que las funciones incorporadas son valores del lenguaje (esto permite pasarlas, almacenarlas en variables, etc.).

Define una función abstracta llamar, que deben implementar todas las primitivas.

llamar recibe una lista de argumentos ***(List<Valor>)*** y devuelve un Valor, que es el resultado de ejecutar la función.

Esto permite que cada función incorporada defina su comportamiento particular de forma flexible.

5. object Incorporadas

¿Qué es Incorporadas?

Este objeto:

Es un singleton Kotlin (object) que agrupa todas las funciones incorporadas.

Tiene un mapa ***Map<String, FuncionIncorporada>***, donde cada clave es el nombre de la función (por ejemplo "+") y el valor es una instancia anónima de una clase que hereda FuncionIncorporada.

Esto permite buscar rápidamente una función incorporada por su nombre (como se escribe en el código Racket).

Cada función incorporada está definida como un objeto anónimo con una implementación específica del método llamar

```
"+" to object : FuncionIncorporada() {
  override fun llamar(args: List<Valor>) =
    Valor.Numero(args.sumOf { (it as Valor.Numero).valor })
}
```

Convierte todos los args a Valor.Numero.

Extrae su valor (.valor) y los suma con sumOf.

Retorna un nuevo Valor.Numero con el resultado.

```
"-" to object : FuncionIncorporada() {
  override fun llamar(args: List<Valor>) =
    Valor.Numero(args.map { (it as Valor.Numero).valor }.reduce(Int::minus))
}
```

Convierte los argumentos a enteros.

Usa reduce(Int::minus) para hacer la resta secuencial

Retorna un nuevo Valor.Numero.

```
"equal?" to object : FuncionIncorporada() {
  override fun llamar(args: List<Valor>) =
    Valor.Booleano(args[0] == args[1])
}
```

Compara directamente los dos primeros argumentos.

Devuelve #t si son iguales, #f si no.

```
"and" to object : FuncionIncorporada() {
  override fun llamar(args: List<Valor>) =
    Valor.Booleano(args.all { (it as Valor.Booleano).valor })
}
```

Convierte cada argumento a Valor.Booleano.

Usa all para verificar si todos son true.

or usa any en lugar de all.

not simplemente niega el único argumento booleano.

```
"<" to object : FuncionIncorporada() {
    override fun llamar(args: List<Valor>) =
        Valor.Booleano((args[0] as Valor.Numero).valor < (args[1] as Valor.Numero).valor)
}
```

Todos los comparadores funcionan igual: acceden a los .valor de los Valor.Numero y devuelven un Valor.Booleano con el resultado.

6. class Entorno

La clase Entorno es una parte fundamental del intérprete Racket en Kotlin, ya que representa el contexto léxico donde se almacenan los valores de variables y funciones. Implementa la interfaz IEntorno, que define las operaciones necesarias para manejar este contexto.

¿Qué representa la clase Entorno?

Entorno es una estructura de datos que:

Guarda asociaciones de nombres (identificadores) con valores (Valor).

Soporta encadenamiento de entornos, permitiendo alcance léxico (lexical scoping).

Se utiliza para evaluar expresiones, definir funciones, y manejar variables en diferentes niveles (local, global, etc.).

```
class Entorno(private val exterior: Entorno? = null) : IEntorno {
  private val valores = mutableMapOf<String, Valor>()
```

valores: es un Map mutable que asocia nombres (como "x", "+", "f") con instancias de Valor.

exterior: es una referencia opcional a un entorno padre (externo). Sirve para permitir búsquedas recursivas cuando una variable no se encuentra en el entorno actual (lexical scoping).

```
override fun definir(nombre: String, valor: Valor) {
  valores[nombre] = valor
}
```

Asigna un valor a un nombre dentro del entorno actual.

Si ya existe una variable con ese nombre, se sobrescribe.

No afecta entornos exteriores.

```
override fun buscar(nombre: String): Valor {
  return valores[nombre] ?: exterior?.buscar(nombre)
    ?: error("Identificador no definido: $nombre")
}
```

Busca un valor asociado al nombre dado.

Primero busca en el entorno actual.

Si no lo encuentra, busca recursivamente en el entorno exterior.

Si no se encuentra en ningún entorno, lanza un error.

override fun extender(): Entorno = Entorno(this)

Crea un nuevo entorno cuyo entorno exterior es el actual.

Se usa al entrar en un nuevo bloque de ejecución, como el cuerpo de una función.

¿Cómo se usa Entorno en el intérprete?

Al iniciar el intérprete, se crea un entorno base (global).

Al definir funciones, se captura el entorno actual como entorno cerrado (clausura).

Al ejecutar una función, se crea un nuevo entorno extendido, con las variables locales (parámetros) definidas.

Cuando se busca una variable, se hace la búsqueda desde el entorno actual hacia los padres.

Ejemplo visual de cómo funciona

(define x 10)

(define (f y) (+ x y))

(f 5) ; debería devolver 15

Esto se traduce internamente a:

En el entorno global:

- x se define como 10
- f se define como una función con el parámetro y, y se captura el entorno global como entorno cerrado

Cuando llamamos a f 5:

- Se crea un nuevo entorno extendido a partir del entorno cerrado
- Se define y = 5 en ese nuevo entorno
- Se evalúa (+ x y):
- x no está en el entorno local, se busca en el padre → 10
- y sí está → 5
- Se devuelve 15

7. class Evaluador

Esta clase implementa la interfaz IEvaluador, y su principal tarea es recorrer y evaluar estructuras sintácticas (Expresion) en un contexto dado (Entorno), produciendo valores (Valor).

¿Qué hace la clase Evaluador?

Evalúa átomos y listas según reglas de evaluación de Racket.

Interpreta construcciones especiales como define, lambda, if.

Evalúa y aplica funciones definidas por el usuario o funciones incorporadas (como +, *, etc.).

Soporta evaluación recursiva, clausuras, y scoping léxico.

```
override fun evaluar(expresion: Expresion, entorno: Entorno): Valor = when (expresion) {
   is Expresion.Atomo -> interpretarAtomo(expresion.valor, entorno)
   is Expresion.Lista -> evaluarLista(expresion.elementos, entorno)
}
```

Esta es la función de entrada principal. Distingue entre dos casos:

Atomo: una constante o nombre.

Lista: una invocación de función o una forma especial (if, define, etc.).

private fun interpretarAtomo(token: String, entorno: Entorno): Valor =

when (token) {

"#t" -> Valor.Booleano(true)

"#f" -> Valor.Booleano(false)

else -> token.toIntOrNull()?.let { Valor.Numero(it) } ?: entorno.buscar(token)

}

Evalúa átomos según su contenido:

"#t" y "#f" se convierten en booleanos.

Si es un número, lo convierte en Valor.Numero.

Si es un símbolo (identificador), lo busca en el entorno.

private fun evaluarLista(elementos: List<Expresion>, entorno: Entorno): Valor

Evalúa expresiones del tipo (f arg1 arg2 ...), que pueden representar:

Una llamada a función, una forma especial (como define, if, lambda), o un bloque anidado.

Pasos:

Verifica que la lista no esté vacía.

Separa el primer elemento (el "operador") y los argumentos.

Si el primer elemento es un átomo, intenta identificar si es una forma especial:

Casos especiales implementados:

"define": define una variable en el entorno actual.

"lambda": crea una función anónima con sus parámetros, cuerpo y entorno cerrado.

"if": evalúa condicionalmente según el valor booleano.

Si no es una forma especial:

Evalúa el primer elemento como función.

Evalúa los argumentos recursivamente.

Llama a aplicarFuncion.

private fun aplicarFuncion(funcion: Valor, argumentos: List<Valor>): Valor

Aplica una función a una lista de valores.

Maneja dos casos:

1. Función definida por el usuario (Valor.Funcion)

    Crea un nuevo entorno extendido (nuevo scope).

    Asigna los argumentos a los parámetros.

    Evalúa el cuerpo de la función en ese entorno.

2. Función incorporada (FuncionIncorporada)

Llama directamente al método llamar.

8. class Interpreter

La clase Interpreter es el componente final del intérprete, el que conecta todos los módulos anteriores para permitir ejecutar código Lisp/Racket desde una cadena de texto, como lo haría un REPL (Read-Eval-Print Loop).

Implementa la interfaz IInterprete, lo cual define su punto de entrada estándar: el método run.

¿Qué hace Interpreter?

Parsea el código fuente usando un ParserSExpresion.

Evalúa cada expresión usando el Evaluador.

Usa un entorno global persistente, que contiene funciones incorporadas (+, -, *, if, etc.).

Devuelve como String el resultado de la última expresión evaluada.

private val parser: IParserExpr = ParserSExpresion()

> Este objeto convierte cadenas como "(+ 1 2)" en estructuras de tipo Expresion (una especie de AST).
>
> Implementa la interfaz IParserExpr, lo que hace intercambiable la implementación del parser.

private val evaluador: IEvaluador = Evaluador()

> Se encarga de recorrer y evaluar el árbol sintáctico (Expresion), transformándolo en un Valor.
>
> Usa el entorno para resolver símbolos, aplicar funciones y manejar estructuras especiales (if, define, etc.).

```
private val entornoGlobal: Entorno = Entorno().apply {
  Incorporadas.funciones.forEach { (nombre, fn) -> definir(nombre, fn) }
}
```

Es el entorno raíz del programa, un mapa de identificadores a valores.

Se inicializa con todas las funciones predefinidas (+, -, etc.) usando la clase Incorporadas.

Se usa como entorno común para todas las expresiones, lo que permite mantener el estado global.

```
override fun run(codigo: String): String {
  val expresiones = parser.parsear(codigo)
  var resultado: Valor = Valor.Indefinido
  for (expresion in expresiones) {
    resultado = evaluador.evaluar(expresion, entornoGlobal)
  }
  return resultado.toString()
}
```

Este método hace tres cosas clave:

Parsear, Evaluar e Imprimir

📘 CLASES Y SU ROL

🔷 1. ParserSExpresion : IParserExpr

Función: Convierte el código fuente (String) en una lista de expresiones (Expresion).

- **Método parsear**: Usa tokenizar y leerDesdeTokens para construir listas o átomos.
- **Implementación:** Basada en un parser recursivo simple de S-expresiones ((a b c)).

🔷 2. Expresion

Clase sellada con dos variantes:

- Atomo(String): Representa identificadores, números o booleanos como texto plano.
- Lista(List<Expresion>): Representa expresiones compuestas como listas de subexpresiones.

Función: Representa la estructura del código fuente, lo que se evalúa.

🔷 3. Valor

Clase sellada que representa los posibles resultados de una evaluación:

- Numero(Int), Booleano(Boolean), Simbolo(String)
- Funcion: Funciones definidas por el usuario (lambda).
- FuncionNativa: Funciones nativas implementadas en Kotlin.
- Indefinido: Representa un valor sin definir (como en define).

Función: Es el "valor en tiempo de ejecución" que resulta de evaluar una expresión.

🔷 4. FuncionIncorporada : Valor

Clase abstracta para funciones internas del lenguaje.

- Tiene un método llamar(args: List<Valor>).
- Se usa para implementar operaciones básicas (+, -, =, and, or, etc.).

Función: Encapsula el comportamiento de funciones primitivas del lenguaje.

🔷 5. Incorporadas

Objeto singleton que contiene todas las funciones internas predefinidas.

- Mapa funciones: Map<String, FuncionIncorporada> que asocia nombres a funciones ("+" to objeto { llamar(...) }).
- Cada función implementa lógica con List<Valor> y retorna un nuevo Valor.

Función: Provee las funciones básicas del lenguaje al entorno global.

🔷 6. Entorno : IEntorno

Función: Mantiene la asociación entre nombres y valores (memoria/ámbito).

- definir: Asigna un valor a un identificador.
- buscar: Recupera valores, buscando recursivamente en entornos padres.
- extender: Crea un entorno nuevo anidado, útil para funciones.

Implementación: Usa un mutableMap y un enlace al entorno exterior para herencia de variables.

🔷 7. Evaluador : IEvaluador

Función: Ejecuta una Expresion dentro de un Entorno, devolviendo un Valor.

- **evalúa átomos:** números, booleanos o variables.
- **evalúa listas:**
    - define: define variables
    - lambda: crea funciones
    - if: evalúa condicionales
    - llamadas a funciones

Función: Es el "corazón" de la ejecución; aplica reglas semánticas.

🔷 8. Interpreter : IInterprete

Función: Conecta todos los componentes.

- Contiene: parser, evaluador, y un entornoGlobal.
- En el método run, parsea, evalúa cada expresión, y devuelve el último resultado.

Implementación: Simple y directa, actúa como REPL o motor principal del intérprete.