Informe taller sobre reconocimiento de patrones

> **Curso:** Fundamentos de Programación Funcional y Concurrente
> **Tema:** Reconocimiento de patrones

Estudiantes:

- Jorge Luis Junior Lasprilla Prada - 2420662
- Andrés Gerardo González Rosero - 2416541

Uso del reconocimiento de patrones

> El reconocimiento de patrones constituye el mecanismo central de definición estructural en la implementación del algoritmo de Huffman en Scala.
> Mediante esta herramienta, las funciones se expresan como casos exhaustivos sobre la forma de los datos, garantizando correspondencia directa entre la semántica matemática del tipo y la sintaxis funcional del programa.

---

peso

Analiza la estructura algebraica del tipo `ArbolH`.
Distingue entre los constructores `Hoja` y `Nodo` para extraer el campo `peso`.
Este uso garantiza la **exhaustividad** del patrón y refleja fielmente la definición inductiva del tipo:

```
arbol match {
  case Hoja(_, p) => p
  case Nodo(_, _, _, p) => p
}
```

---

cars

Usa reconocimiento de patrones sobre `ArbolH` para extraer la lista de caracteres:

```
arbol match {
  case Hoja(c, _) => List(c)
  case Nodo(_, _, cs, _) => cs
}
```

El patrón determina la forma estructural del árbol y permite definir la función de forma declarativa, sin condicionales ni accesos posicionales.

ocurrencias

Aplica reconocimiento de patrones sobre listas de caracteres.
Utiliza el esquema estructural `h :: t` para representar la recursión sobre listas:

```
xs match {
  case Nil => Nil
  case h :: t => ...
}
```

Esta forma constituye la base inductiva de la función, expresando la descomposición natural de una lista en cabeza y cola.

---

listaUnitaria

Distingue entre listas vacías, unitarias y de longitud mayor mediante patrones estructurales:

```
arboles match {
  case _ :: Nil => true
  case _ => false
}
```

La coincidencia `x :: Nil` define formalmente la unicidad estructural de un elemento en la lista.

---

combinar

Descompone la lista de árboles de Huffman en sus dos primeros elementos y el resto:

```
arboles match {
  case a1 :: a2 :: resto => ...
  case _ => arboles
}
```

El patrón garantiza la aplicación binaria correcta y preserva la estructura inductiva de la lista.

---

hastaQue

No contiene `match` explícito, pero su comportamiento se apoya en funciones (`combinar`, `listaUnitaria`) que sí lo utilizan internamente.

Su corrección depende de la **recursión estructural** sobre listas derivada de dichas funciones.

---

crearArbolDeHuffman

Compone funciones que operan mediante reconocimiento de patrones (`ocurrencias`, `listaDeHojasOrdenadas`, `combinar`).
El uso de patrones se manifiesta indirectamente a través de la composición funcional que conserva la estructura de los datos de entrada.

---

decodificar

Evalúa la estructura de un árbol de Huffman y una lista de bits mediante coincidencia estructural:

```
actual match {
  case Hoja(c, _) => ...
  case Nodo(izq, der, _, _) => ...
}
```

Cada patrón corresponde a una cláusula de la definición recursiva de decodificación.
Esta correspondencia garantiza la **totalidad** de la función y la correcta interpretación de los bits según la estructura del árbol.

---

codificar

Utiliza patrones sobre `ArbolH` para determinar el recorrido según el carácter buscado:

```
arbol match {
  case Hoja(ch, _) if ch == c => Nil
  case Nodo(izq, der, _, _)   => ...
}
```

El patrón con guardia (`if ch == c`) refuerza la **selectividad semántica** y asegura que cada caso estructural se evalúe correctamente.

---

codigoEnBits

Aplica reconocimiento de patrones sobre listas de pares `(Char, List[Bit])` para realizar búsquedas recursivas:

```
tabla match {
  case (c, bits) :: tail => ...
```

```
  case Nil => Nil
}
```

El patrón expresa de forma declarativa la descomposición de una lista de asociaciones, equivalente a una búsqueda inductiva sobre pares ordenados.

---

mezclarTablasDeCodigos

Usa **desestructuración implícita** en expresiones lambda, donde cada par `(c, bits)` se descompone en sus componentes:

```
a.map { case (c, bits) => (c, 0 :: bits) }
```

El patrón actúa como una forma declarativa de acceso estructural, eliminando la necesidad de índices o selecciones posicionales.

---

convertir

Descompone recursivamente el tipo `ArbolH` en sus constructores, representando de forma explícita la definición inductiva del árbol:

```
arbol match {
  case Hoja(c, _) => ...
  case Nodo(izq, der, _, _) => ...
}
```

Cada rama del patrón corresponde a un caso del tipo algebraico, asegurando la exhaustividad y terminación de la función.

---

codificarRapido

No emplea `match` directamente, pero depende de las funciones `convertir` y `codigoEnBits`, las cuales sí aplican reconocimiento de patrones sobre estructuras inductivas.
Por composición, su comportamiento hereda la **corrección estructural** de dichas funciones.

Informe de Corrección

> A continuación se presentan las argumentaciones formales de corrección de las funciones
> `ocurrencias`, `combinar`, `codificar`, `decodificar` y `codificarRapido`.

Función `ocurrencias`

Sea

$$f : \text{List}[\text{Char}] \to \text{List}[(\text{Char}, \text{Int})]$$

la función matemática que asocia a cada carácter (c) de una lista (L) su número de apariciones (n_c).
Sea (P_f) la función implementada en Scala:

$$P_f(L) = \begin{cases} [] & \text{si } L = [] \\ \texttt{actualizar}(x, P_f(xs)) & \text{si } L = x :: xs \end{cases}$$

Objetivo: Demostrar que

$$\forall L \in \text{List}[\text{Char}],\ P_f(L) = f(L)$$

Prueba por inducción estructural sobre la lista (L):

- **Caso base:**
  Si (L = []), entonces (P_f([]) = []) y (f([]) = []).
  Por lo tanto, (P_f([]) = f([])).
- **Paso inductivo:**
  Supóngase que para una lista (xs), (P_f(xs) = f(xs)).
  Sea (L = x :: xs). Entonces:

  $$P_f(L) = \texttt{actualizar}(x, P_f(xs))$$

  Aplicando la hipótesis inductiva, (P_f(xs) = f(xs)).
  Por la definición de \texttt{actualizar}, el resultado incrementa en uno la frecuencia de (x), de modo que:

  $$P_f(L) = f(x :: xs)$$

Por lo tanto, (P_f(L) = f(L)) para toda lista (L).
La función es **correcta**.

Función `combinar`

Sea

$$f : \text{List}[\text{ArbolH}] \to \text{List}[\text{ArbolH}]$$

la función que combina los dos árboles de menor peso en una lista ordenada ascendentemente.
Su implementación Scala (P_f) es:

$$P_f(L) = \begin{cases} L & \text{si } |L| < 2 \\ \texttt{insertarOrdenado}(\texttt{hacerNodoArbolH}(a_1, a_2), R) & \text{si } L = a_1 :: a_2 :: R \end{cases}$$

Propiedad a demostrar:

$$\forall L,\ P_f(L) = f(L)$$

Demostración por inducción sobre (n = |L|):

- **Caso base:**
  Si (n < 2), no hay árboles que combinar.
  (P_f(L) = L = f(L)).
- **Paso inductivo:**
  Supóngase que (P_f(L) = f(L)) para toda lista de tamaño (n).
  Sea (L = a_1 :: a_2 :: R) de tamaño (n+1). Entonces:

  $$P_f(L) = \texttt{insertarOrdenado}(\texttt{hacerNodoArbolH}(a_1, a_2), R)$$

  Como (p(\texttt{hacerNodoArbolH}(a_1,a_2)) = p(a_1) + p(a_2) \geq p(a_1)),
  y (R) está ordenado, al insertar el nuevo nodo mediante \texttt{insertarOrdenado} se mantiene la propiedad de orden.

Por tanto, (P_f(L) = f(L)) y la función preserva la invariante de orden y corrección estructural del algoritmo de Huffman.

Función `codificar`

Sea

$$f : \text{List}[\text{Char}] \to \text{List}[\text{Bit}]$$

la función que asigna a cada carácter su secuencia binaria según un árbol de Huffman (A).
Su implementación Scala (P_f) es:

$$P_f(A, M) = \begin{cases} [] & \text{si } M = [] \\ \texttt{codChar}(A, c) \ ++ \ P_f(A, cs) & \text{si } M = c :: cs \end{cases}$$

Teorema de corrección:

$$\forall M,\ P_f(A, M) = f(M)$$

Demostración por inducción sobre la lista (`M`):

- **Caso base:**
  (M = [] \implies P_f(A, M) = [] = f([])).
- **Paso inductivo:**
  Sea (M = c :: cs). Por definición:

  $$P_f(A, M) = \texttt{codChar}(A, c) + + P_f(A, cs)$$

  Por hipótesis inductiva, (P_f(A, cs) = f(cs)).
  Además, (\texttt{codChar}(A, c) = f(c)) por definición del árbol de Huffman.
  Luego:

$$P_f(A, M) = f(c) + +f(cs) = f(M)$$

Se cumple que (P_f(A, M)) codifica correctamente cualquier mensaje según el árbol (A).

Función `decodificar`

Sea

$$f : \text{List}[\text{Bit}] \to \text{List}[\text{Char}]$$

la función que reconstruye el mensaje a partir de su codificación binaria y el árbol (A). Su implementación (P_f) es:

$$P_f(A, B) = \begin{cases} [] & \text{si } B = [] \\ c :: P_f(A, B') & \text{si la lectura de bits en } B \text{ lleva a la hoja } c \text{ con resto } B' \end{cases}$$

Objetivo:

$$\forall B,\ P_f(A, B) = f(B)$$

Demostración por inducción sobre la lista de bits (B):

- **Caso base:**
  (B = [] \implies P_f(A, B) = [] = f([])).
- **Paso inductivo:**
  Sea (B = b :: bs).
  El algoritmo sigue el árbol según el valor de cada bit: (0) implica desplazamiento al subárbol izquierdo y (1) al derecho.
  Al llegar a una hoja con símbolo (c), se consume la porción correspondiente de bits y se continúa sobre el resto (B').

  Por hipótesis inductiva, (P_f(A, B') = f(B')), y por la propiedad de prefijos del código de Huffman, la lectura es determinista y completa.
  Entonces:

  $$P_f(A, B) = c :: f(B') = f(B)$$

Por tanto, la función `decodificar` es la inversa de `codificar` respecto al mismo árbol (A).

Función `codificarRapido`

Sea

$$f : \text{List}[\text{Char}] \to \text{List}[\text{Bit}]$$

la función que usa una tabla de códigos (T) generada por \texttt{convertir(A)}:

$$f(M) = \bigcup_{c \in M} T[c]$$

La implementación Scala (P_f) se define como:

$$P_f(A, M) = \texttt{codigoEnBits}(T, M) \quad \text{donde} \quad T = \texttt{convertir}(A)$$

Teorema de equivalencia:

$$\forall M,\ P_f(A, M) = \texttt{codificar}(A, M)$$

Demostración:

La función \texttt{convertir(A)} recorre el árbol (A) y asigna a cada carácter (c) su secuencia binaria exacta (\texttt{codChar}(A, c)).
Por definición de (T):

$$\forall c \in \Sigma,\ T[c] = \texttt{codChar}(A, c)$$

Entonces:

$$P_f(A, M) = \bigcup_{c \in M} T[c] = \bigcup_{c \in M} \texttt{codChar}(A, c) = \texttt{codificar}(A, M)$$

Por tanto, (P_f) es funcionalmente equivalente a \texttt{codificar}, pero con menor complejidad temporal gracias al acceso directo mediante la tabla (T).

Casos de Prueba

> A continuación se documentan 5 casos de prueba por cada función definida en `package.scala` (package object `Huffman`). Las tablas muestran entradas (simplificadas), una breve descripción y el resultado esperado. Se ejecutan en `Pruebas.sc` mediante aserciones.

peso

Caso	Entrada	Descripción	Resultado Esperado
1	Hoja('a',5)	Peso de hoja simple	5
2	Hoja('b',2)	Peso de hoja simple	2
3	Nodo(Hoja('a',5),Hoja('b',2))	Suma de pesos hijos	7
4	Nodo(Nodo(...,7),Hoja('a',5))	Suma recursiva 7+5	12
5	arbolBase (texto "abbcccdddd")	Peso total = longitud	10

cars

Caso	Entrada	Descripción	Resultado Esperado
1	Hoja('z',1)	Lista de chars de hoja	List('z')
2	Nodo(Hoja('a',1),Hoja('b',1))	Unión chars a y b	List('a','b') (orden no crítico)
3	Nodo(prev,Hoja('c',1))	Inclusión de nuevo char	Contiene 'c'
4	arbolBase	Chars del árbol	Distinct de texto base
5	arbolBase	Sin duplicados	size == distinct.size

hacerNodoArbolH

Caso	Entrada	Descripción	Resultado
1	(Hoja x3, Hoja y4)	Peso suma	7
2	(Hoja x3, Hoja y4)	Chars combinados	x,y
3	Nodo(xy, Hoja z1)	Peso acumulado	8
4	Nodo(xy, Hoja z1)	Contiene z	true
5	Nodo(xy, Hoja z1)	Total chars	3

cadenaAListA

Caso	Entrada	Descripción	Resultado
1	""	Cadena vacía	Nil
2	"a"	Un carácter	List('a')
3	"ab"	Dos caracteres	List('a','b')
4	"aba"	Mantiene duplicados	List('a','b','a')
5	textoBase	Conversión correcta	coincide

ocurrencias

Caso	Entrada	Descripción	Resultado
1	Nil	Lista vacía	Nil
2	List('a')	Un elemento	('a',1)
3	a,a,b	Frecuencias parciales	a->2, b->1
4	textoBase	Frecuencias completas	a1 b2 c3 d4
5	textoBase	Orden aparición	a,b,c,d

listaDeHojasOrdenadas

Caso	Entrada	Descripción	Resultado
1	(b3,a1,c2)	Orden ascendente	a,c,b
2	(b3,a1,c2)	Pesos orden asc	1,2,3
3	textoBase	Número de hojas	4
4	textoBase	Primera hoja	'a'
5	textoBase	Última hoja	'd'

listaUnitaria

Caso	Entrada	Descripción	Resultado
1	Nil	No unitaria	false
2	[Hoja]	Unitaria	true
3	[Hoja,Hoja]	No unitaria	false
4	[3 hojas]	No unitaria	false
5	[arbolBase]	Unitaria	true

combinar

Caso	Entrada	Descripción	Resultado
1	3 hojas (1,2,3)	Reduce tamaño	2 elementos
2	post-comb	Conserva suma pesos	6
3	2 hojas resultantes	Reduce a 1	1 elemento
4	lista unitaria	Idempotente	Igual
5	Nil	Caso vacío	Nil

hastaQue

Caso	Entrada	Descripción	Resultado
1	hojas textoBase	Termina en unitaria	size=1
2	lista unitaria	Idempotente	misma lista
3	Nil + cond vacía	Retorna Nil	Nil
4	2 hojas	Una combinación	size=1
5	2 hojas	Conserva suma pesos	igual suma

crearArbolDeHuffman

Caso	Entrada	Descripción	Resultado
1	textoBase	Peso total	10
2	textoBase	Conjunto chars	{a,b,c,d}
3	"aaaa"	Peso acumulado	4
4	"aaaa"	Chars únicos	a
5	"ab"	Peso	2

decodificar

Caso	Entrada	Descripción	Resultado
1	bits(textoBase)	Roundtrip	textoBase
2	bits(duplicado)	Tamaño doble	2 * len
3	bits(textoBase)	No vacío	true
4	bits("aa" árbol único)	Caso hoja única	"aa"
5	bits rápidos	Equivalente	textoBase

codificar

Caso	Entrada	Descripción	Resultado
1	textoBase	No vacío	bits != Nil
2	textoBase	Roundtrip	textoBase
3	duplicado	Solo 0/1	true
4	"aa" árbol único	Código vacío	Nil
5	ddc	Roundtrip parcial	"ddc"

codigoEnBits

Caso	Entrada	Descripción	Resultado
1	tablaBase	Cobertura	todos chars
2	'a','b'	Códigos distintos	true
3	'a'	Determinista	igual
4	'#'	Ausente => Nil	Nil
5	'a'	Longitud válida	>=0

mezclarTablasDeCodigos

Caso	Entrada	Descripción	Resultado
1	ta,tb	Tamaño combinado	4
2	ta	Prefijo 0	true
3	tb	Prefijo 1	true
4	mezcla	Sin códigos vacíos	true
5	mezcla	Códigos únicos	true

convertir

Caso	Entrada	Descripción	Resultado
1	arbolBase	Todos chars presentes	true
2	arbolBase	Algún código no vacío	true
3	arbolBase	Códigos únicos	true
4	árbol único	Código vacío	('a',Nil)
5	arbolBase	Media longitud razonable	<= num chars

codificarRapido

Caso	Entrada	Descripción	Resultado
1	textoBase	Igual a codificar	true
2	ddc	Decodifica parcial	"ddc"
3	"aa" árbol único	Lista vacía	Nil
4	duplicado	Roundtrip duplicado	2 * texto
5	textoBase	Misma longitud que codificar	true

Conclusiones

Este taller nos permitió entender mucho mejor cómo el reconocimiento de patrones no es solo una característica sintáctica de Scala, sino una forma de razonar directamente sobre la estructura de los datos (listas, árboles, etc.) desde la lógica del propio lenguaje.
Al trabajar con las funciones del algoritmo de Huffman, pudimos ver que cada uso de `match` o de desestructuración no es un detalle técnico, sino la representación práctica del principio de **inducción estructural** (algo que normalmente se ve en teoría, pero aquí se aplica de forma muy concreta).

Durante el proceso, noté que definir funciones como `ocurrencias`, `combinar`, `codificar` o `decodificar` con patrones hace que el código sea más predecible y más fácil de verificar formalmente. Cada caso del patrón cubre una posibilidad exacta del tipo de dato, lo que evita errores lógicos y garantiza exhaustividad sin tener que depender de condicionales o comprobaciones adicionales.

También me pareció importante ver cómo el reconocimiento de patrones puede ser **explícito** (cuando uso directamente un `match`) o **implícito** (cuando desestructuro una tupla, una lista o aplico un `map` con un `case` dentro). En ambos casos, la idea es la misma: expresar el comportamiento del programa siguiendo la forma del dato, no forzando la estructura desde fuera.

Finalmente, puedo concluir que este enfoque no solo mejora la legibilidad, sino que también facilita el razonamiento formal del código —uno entiende *por qué* la función es correcta, no solo que “funciona”.
En definitiva, el taller refuerza la idea de que la programación funcional no se trata solo de “no usar bucles ni mutabilidad”, sino de pensar el código como una extensión lógica y estructurada de las definiciones matemáticas sobre las que se construye.