
# Taller 3: Reconocimiento de Patrones

Angie Natalia Córdoba Collazos 2124366 - 3743
Wilson Andres Mosquera Zapata 2182116 - 3743

Octubre 2023

# Informe del taller

En este taller estaremos trabajando la tecnica de reconocimiento de patrones, por medio del problema planteado en el curso: maniobra de trenes en una estación utilizando la programación funcional en el lenguaje de Scala. La estación se compone de tres trayectos: el trayecto principal y los trayectos auxiliares: uno y dos. Ahora, durante el taller, cada configuración en la estación de maniobras se describira como un estado y un movimiento especifico definirá como los vagones se trasladan de un trayecto a otro. El objetivo final será encontrar una secuencia de movimientos que permitirá transformar la configuración del tren desde el trayecto principal en otro deseado.

| Función | Reconocimiento de Patrones | Argumento |
|---|---|---|
| aplicarMovimiento | Si | Se utilizan  patrones para identificar el tipo y valor de cada uno de los movimientos. Realiza las operaciones que corresponden en función de estos patrones. |
| aplicarMovimientos | Si | En cada iteración, se reconoce el patrón del siguiente movimiento en la lista y se llama recursivamente a la función para aplicar el movimiento al estado actual |
| definirManiobra | Si | Se utiliza el reconocimiento de patrones para desglosar los pares de vagones de los trenes t1 y t2. Esto posibilita la identificación y gestión eficiente de los vagones que deben ser desplazados a sus posiciones correctas en el tren objetivo, facilitando así la creación de la maniobra precisa para alcanzar la configuración deseada. |

## Informe de Procesos:

- **Función aplicarMovimiento:**

```java
//1.2.1 APLICAR MOVIMIENTO//

def aplicarMovimiento(e: Estado, m: Movimiento): Estado = m match {
    case Uno(n) if n > 0 =>
      val (take, remaining) = e._1.splitAt(e._1.length - n)
      ((take, e._2 ++ remaining, e._3))

    case Uno(n) if n < 0 =>
      val (take, remaining) = e._2.splitAt(n.abs)
      ((e._1 ++ take, remaining, e._3))
```

```
    case Dos(n) if n > 0 =>
      val (take, remaining) = e._1.splitAt(n)
      ((remaining, e._2, take ++ e._3))

    case Dos(n) if n < 0 =>
      val (take, remaining) = e._3.splitAt(n.abs)
      ((e._1 ++ take, e._2, remaining))

    case _ => e
  }
```

- **Función aplicarMovimientos:**

Java

```
//1.2.1 APLICAR MOVIMIENTOS//

def aplicarMovimientos(estadoInicial: Estado, movimientos: Maniobra):
List[Estado] = {
  def generarListaEstados(estado: Estado, movimientosRestantes:
Maniobra): List[Estado] = movimientosRestantes match {
    case Nil => Nil
    case movimiento :: rest =>
      val nuevoEstado = aplicarMovimiento(estado, movimiento)
      nuevoEstado :: generarListaEstados(nuevoEstado, rest)
  }
  estadoInicial :: generarListaEstados(estadoInicial, movimientos)
}
```

- **Función definirManiobra:**

Java

```
//1.2.1 DEFINIR MANIOBRAS//

def definirManiobra(t1: Tren, t2: Tren): Maniobra = {
    t1.zip(t2).flatMap {
      case (vagonT1, vagonT2) =>
        if (vagonT1 == vagonT2) {
          Nil
```

```
        } else {
          val movesToTarget = if (t2.indexOf(vagonT1) != -1)
t2.indexOf(vagonT1) - t1.indexOf(vagonT1) else 0
          Uno(movesToTarget) :: Uno(-1) :: Nil
        }
    }
  }
```

## Informe de Correcciones:

- **Función aplicarMovimiento:**

```
Java
//1.2.1 APLICAR MOVIMIENTO//

def aplicarMovimiento(e: Estado, m: Movimiento): Estado = m match {
  case Uno(n) if n > 0 =>
    val (take, remaining) = e._1.splitAt(e._1.length - n)
    ((take, e._2 ++ remaining, e._3))
  case Uno(n) if n < 0 =>
    val (take, remaining) = e._2.splitAt(n.abs)
    ((e._1 ++ take, remaining, e._3))
  case Dos(n) if n > 0 =>
    val (take, remaining) = e._1.splitAt(n)
    ((remaining, e._2, take ++ e._3))
  case Dos(n) if n < 0 =>
    val (take, remaining) = e._3.splitAt(n.abs)
    ((e._1 ++ take, e._2, remaining))
  case _ => e
}
```

La función aplicarMovimiento utiliza los siguientes casos:

➔ Caso **Uno(n)** cuando **n > 0** donde se deben mover **n** vagones del trayecto principal al trayecto uno y mantenemos los vagones restantes en el trayecto principal.

- Formalmente definimos TrenPrincipal, TrenUno y trenDos de la siguiente manera:

$$TrenPrinicipal' = TrenPrincipal - (n\ últimos\ vagones)$$

$$TrenUno' = TrenUno ++ (n\ últimos\ vagones)$$

$$TrenDos' = TrenDos$$

- El movimiento Uno(n) con n > 0 mueve n vagones del tren principal al tren uno, mnteniendo el tren dos sin cambios. Esto se logra tomando los primeros len(TrenPrincipal) - n vagones del tren principal y agregándolos al tren uno, mientras que el tren principal mantiene los últimos n vagones. El tren dos no se modifica. Por lo tanto, el nuevo estado e' satisface las fórmulas.

➔ Caso **Uno(n)** cuando **n < 0** donde se deben mover n vagones del trayecto uno al trayecto principal y mantenemos los vagones restantes en el trayecto uno.

- Formalmente definimos TrenPrincipal, TrenUno y TrenDos de la siguiente manera:

$$TrenPrinicipal' = TrenPrincipal \ ++ \ (n\ últimos\ vagones)$$

$$TrenUno' = TrenUno \ - \ (n\ últimos\ vagones)$$

$$TrenDos' = TrenDos$$

- El movimiento Uno(n) con n < 0 mueve n.abs vagones del tren uno al tren principal manteniendo el tren dos sin cambios. Esto se logra tomando los primeros n.abs vagones del tren uno y agregándolos al tren principal, mientras que el tren uno mantiene los vagones desde n.abs hasta el final. El tren dos no se modifica. Por lo tanto, el nuevo estado e' satisace las formulas.

➔ Caso **Dos(n)** cuando **n > 0** donde se deben mover n vagones del trayecto principal al trayecto dos y mantenemos los vagones restantes en el trayecto principal.

- Formalmente definimos TrenPrincipal, TrenUno y TrenDos de la siguiente manera:

$$TrenPrinicipal' = TrenPrincipal \ - \ (n\ primeros\ vagones)$$

$$TrenUno' = TrenUno$$

$$TrenDos' = TrenDos \ ++ \ (n\ primeros\ vagones)$$

- El movimiento Dos(n) con n > 0 mueve n vagones del tren principal al tren dos, manteniendo el tren uno sin cambios. Esto se logra tomando los primeros n vagones del tren principal y agregándolos al tren dos mientras que el tren principal mantiene los vagones desde n hasta el final. Entre todo esto el tren uno no se modifica. Por lo tanto, el nuevo estado e' satisface las fórmulas.

➔ Caso Dos(n) cuando n < 0 donde se deben mover n vagones del trayecto dos al trayecto principal y mantenemos los vagones restantes en el trayecto dos.

- Formalmente definimos TrenPrincipal, TrenUno y TrenDos de la siguiente manera:

$$TrenPrinicipal' = TrenPrincipal \; ++ \; (n \; últimos \; vagones)$$

$$TrenUno' = TrenUno$$

$$TrenDos' = TrenDos \; - \; (n \; últimos \; vagones)$$

- El movimiento Dos(n) con n < 0 mueve n.abs vagones del tren dos al tren principal, manteniendo el tren uno sin cambios. Esto se logra tomando los primeros n.abs vagones del tren dos y agregándolos al tren principal, mientras que el tren dos mantiene los vagones desde n.abs hasta el final. El tren uno no se modifica. Por lo tanto, podemos decir que el nuevo estado e' satisface las formulas.

➔ Caso _ (cualquier otro movimiento), en este caso el estado se mantiene igual. No se realiza ningún cambio en los trenes.

- Por lo que formalmente tendríamos:

$$TrenPrinicipal' = TrenPrincipal$$

$$TrenUno' = TrenUno$$

$$TrenDos' = TrenDos$$

- La función aplicarMovimiento no realiza ningún cambio en el estado, por lo que tendríamos que e' = e.

## Casos de prueba

Ejecución:

```
1       import ManiobrasTrenes._
2
3       val e1 = (List('a', 'b', 'c', 'd'), Nil, Nil)
4       val e2 = aplicarMovimiento(e1, Uno(2))
5       val e3 = aplicarMovimiento(e2, Dos(3))
6       val e4 = aplicarMovimiento(e3, Dos(-1))
7       val e5 = aplicarMovimiento(e4, Uno(-2))
8  →    val e6 = aplicarMovimiento(e2, Uno(7))
```

Resultado:

```
import ManiobrasTrenes._

val e1: (List[Char], collection.immutable.Nil.type, collection.immutable.Nil.type) = (List(a, b, c, d),List(),List())
val e2: ManiobrasTrenes.Estado = (List(a, b),List(c, d),List())
val e3: ManiobrasTrenes.Estado = (List(),List(c, d),List(a, b))
val e4: ManiobrasTrenes.Estado = (List(b),List(c, d),List(a))
val e5: ManiobrasTrenes.Estado = (List(c, d),List(b),List(a))
val e6: ManiobrasTrenes.Estado = (List(c, d),List(a, b),List())
```

Agregamos el caso de prueba e6, donde aplicamos Uno(7) al estado e2. Este estado ya tiene dos vagones en el trayecto uno (c y d) y dos vagones en el tren principal (a y b). Cuando aplicamos Uno(7) para mover 7 vagones del tren principal al trayecto uno, esto supera al número de vagones disponible en el tren principal. Como resultado vemos que se deben mover todos los vagones del tren principal al trayecto uno. Este resultado lo vemos correctamente en las fotos adjuntas.

- **Función aplicarMovimientos:**

```
Java
//1.2.2 APLICAR MOVIMIENTOS//
def aplicarMovimientos(estadoInicial: Estado, movimientos: Maniobra):
List[Estado] = {
  def generarListaEstados(estado: Estado, movimientosRestantes:
Maniobra): List[Estado] = movimientosRestantes match {
    case Nil => Nil
    case movimiento :: rest =>
      val nuevoEstado = aplicarMovimiento(estado, movimiento)
      nuevoEstado :: generarListaEstados(nuevoEstado, rest)
  }
  estadoInicial :: generarListaEstados(estadoInicial, movimientos)
}
```

Tenemos:

- ➔ e como estado inicial, donde $e = (t_1, t_2, t_3)$, donde $t_1, t_2, t_3$ son listas de vagones.
- ➔ movs como la secuencia de movimientos, $movs = [m_1, m_2, \ldots, m_k]$, donde $mi$ es el i-ésimo movimiento.
- ➔ res como el resultado de aplicar aplicarMovimientos(e,movs), que es una lista de estados.

**Caso Base:**
Cuando la lista de movimientos está vacía,, movs = [], no se aplican movimientos, por lo que el estado resultante es el mismo que el estado inicial e. Entonces, tenemos:

$$res = [e_0]$$

Esto demuestra que la función es correcta para el caso base.

**Hipotesis de Inducción:**
Supongamos que la función aplicarMovimientos es correcta para una secuencia de movimientos de longitud k, es decir, cuando $mov$ es $[m_1, m_2, \ldots, m_k]$. Esto significa que la lista de estados producida por aplicarMovimientos $(e_0, [m_1, m_2, \ldots m_k])$ es $[e_1, e_2, \ldots e_k]$.

**Paso de Inducción:**
Ahora, demostraremos que la función aplicarMovimientos también es correcta para una secuencia de movimientos de longitud $k + 1$, es decir, cuando $movs$ es $[m_1, m_2, \ldots m_{k+1}]$.

- ➔ Usamos la hipótesis de inducción para asumir que aplicarMovimientos $(e, [m_1, m_2, \ldots m_k])$ produce una lista de estados $[e, e_1, e_2, \ldots e_k]$.
- ➔ Luego, aplicamos el movimiento $m_{k+1}$ al último estado de esa lista, $e_k$, utilizando aplicarMovimientos $(e_k, m_{k+1})$ para obtener un nuevo estado $e_{k+1}$.
- ➔ Añadimos $e_{k+1}$ a la lista existente de estados $[e, e_1, e_2, \ldots e_k]$ para obtener la nueva lista $[e, e_1, e_2, \ldots, e_k, e_{k+1}]$.
- ➔ Entonces la lista de estados producidas por aplicarMovimiento $(e, [m_1, m_2, \ldots m_k]), \ldots, [m_k, m_{k+1}]$ es $(e, [e, e_1, e_2, \ldots e_k, e_{k+1}])$.

Esto demuestra que esta lista representa los estados a lo largo de la secuencia de movimientos desde $e$ hasta $e_{k+1}$, cumpliendo así con el propósito de la función aplicarMovimientos.

**Casos de prueba**

Los casos de prueba usados para la verificación de esta función fueron los siguientes:

```
val e = (List('a','b'),List('c'), List('d'))
aplicarMovimientos(e, List(Uno(1), Dos(1), Uno(-2)))

val m1 = (List(1, 2, 3, 4, 5), Nil, Nil)
aplicarMovimientos(m1, List(Uno(2), Dos(3), Dos(-1), Uno(-2), Dos(-1)))
aplicarMovimientos(m1, List(Uno(5), Uno(-1), Dos(0), Dos(4), Uno(3)))
aplicarMovimientos(m1, List(Dos(3), Dos(-2), Uno(1), Uno(4), Dos(2)))
aplicarMovimientos(m1, List(Dos(0), Uno(6), Dos(2), Dos(-1), Uno(-1)))
```

Al usar los anteriores casos de prueba obtuvimos como salidas lo siguiente:

```
val e: (List[Char], List[Char], List[Char]) = (List(a, b),List(c),List(d))
val l1: (List[Int], collection.immutable.Nil.type, collection.immutable.Nil.type) = (List(1, 2, 3, 4, 5),List(),List())

val res0: List[ManiobrasTrenes.Estado] = List((List(a, b),List(c),List(d)), (List(a),List(c, b),List(d)), (List(a),List(c, b),List(d))
val res1: List[ManiobrasTrenes.Estado] = List((List(1, 2, 3, 4, 5),List(),List()), (List(1, 2, 3),List(4, 5),List()), (List(1, 2, 3),L
val res2: List[ManiobrasTrenes.Estado] = List((List(1, 2, 3, 4, 5),List(),List()), (List(1, 2),List(3, 4, 5),List()), (List(1, 2, 3, 4
val res3: List[ManiobrasTrenes.Estado] = List((List(1, 2, 3, 4, 5),List(),List()), (List(5),List(),List(1, 2, 3, 4)), (List(5, 1, 2),L
```

Con estas salidas comprobamos el correcto funcionamiento de la función anterior ya que sus salidas fueron las que se esperaban.

- **Función definirManiobra:**

Java

```
//1.2.1 DEFINIR MANIOBRAS//

def definirManiobra(t1: Tren, t2: Tren): Maniobra = {
    t1.zip(t2).flatMap {
      case (vagonT1, vagonT2) =>
        if (vagonT1 == vagonT2) {
          Nil
        } else {
          val movesToTarget = if (t2.indexOf(vagonT1) != -1)
t2.indexOf(vagonT1) - t1.indexOf(vagonT1) else 0
          Uno(movesToTarget) :: Uno(-1) :: Nil
        }
    }
```

- En el escenario de vagones idénticos:

Para demostrar que cuando los vagones son idénticos, la función produce una lista vacía de maniobras (Nil), formularemos la hipótesis que deseamos validar:

**Hipótesis (H1):** Si los vagones son idénticos, entonces la función resulta en una lista vacía de maniobras: *definirManiobra(vagonT1, vagonT2) = Nil.*

**Demostración:**
Cuando *vagonT1* es igual a *vagonT2*, no se requiere el desplazamiento de ningún vagón, lo que resulta en una lista vacía de maniobras.

Por lo tanto, la *Hipótesis H1* es confirmada.

- Escenario de Transferencia de Vagón de t1 a t2:

Con el propósito de demostrar que la función genera las maniobras apropiadas para mover un vagón de *t1* a *t2*, plantearemos la siguiente hipótesis:

**Hipótesis (H2):** Si un vagón en *t1* no ocupa la misma posición en *t2*, entonces la función generará la maniobra correspondiente para su traslado.

**Demostración**:
Supongamos que *vagonT1* no se encuentra en la misma posición en *t2*, es decir, *t2.indexOf(vagonT1) != -1 y t2.indexOf(vagonT1) != t1.indexOf(vagonT1).*

En tal caso, la función generará la maniobra *Uno(movesToTarget) :: Uno(-1) :: Nil*, donde movesToTarget representa la cantidad de movimientos necesarios para trasladar *vagonT1* de *t1* a *t2*.

Por lo tanto, la *Hipótesis H2* es confirmada.

- Escenario de Movimiento Nulo:

Para demostrar que, cuando no se requiere movimiento, la función arroja una lista vacía de maniobras (Nil), presentaremos la hipótesis que deseamos validar:

**Hipótesis (H3):** Si no se precisa ningún movimiento, entonces la función da como resultado una lista vacía de maniobras: *definirManiobra(t1, t2) = Nil.*

**Demostración:**
Si no existen diferencias entre los vagones de *t1* y *t2*, no es necesario efectuar movimiento alguno, conduciendo a la generación de una lista vacía de movimientos.

Por lo tanto, la *Hipótesis H3* es confirmada.

A partir de estas hipótesis y demostraciones, se corrobora que la función *"definirManiobra"* cumple con los escenarios examinados y produce los movimientos apropiados en cada circunstancia.

**Casos de prueba:**

Para esta función designamos las siguientes entradas para la comprobación de la función:

```
definirManiobra(List('a', 'b', 'c', 'd') , List('d', 'b', 'c', 'a'))
definirManiobra(List(1, 3, 5, 7, 9) , List(2, 4, 6, 8, 10))
definirManiobra(List(6, 5, 4, 3, 2) , List(2, 3, 4, 5, 6))
```

Las salidas que se obtuvieron fueron las siguientes:

```
val res4: ManiobrasTrenes.Maniobra = List(Uno(3), Uno(-1), Uno(-3), Uno(-1))
val res5: ManiobrasTrenes.Maniobra = List(Uno(0), Uno(-1), Uno(0), Uno(-1), Uno(0), Uno(-1), Uno(0), Uno(-1), Uno(0), Uno(-1))
val res6: ManiobrasTrenes.Maniobra = List(Uno(4), Uno(-1), Uno(2), Uno(-1), Uno(-2), Uno(-1), Uno(-4), Uno(-1))
```

## Conclusión

En este taller se realizó un código funcional usando la técnica de reconocimiento de patrones para la resolución de un problema de maniobra de trenes en una estación de maniobras en el cual el objetivo era reordenar el tren dado con la ayuda de estaciones auxiliares de forma tal que se obtuviera el tren deseado. El proyecto consistía en encontrar una secuencia de movimientos que convirtieran el tren del trayecto principal, en otra forma del tren sobre el trayecto principal.
Después de abordar las funciones utilizando la técnica de reconocimiento de patrones, hemos llegado a apreciar su utilidad y eficacia. Al aplicar el reconocimiento de patrones, es posible lograr funcionalidades y comportamientos de manera concisa que, de otra manera, requerirían la implementación de múltiples funciones o largas líneas de código. Esta técnica si se utiliza de manera correcta es una herramienta poderosa y eficaz.