Taller 3: Reconocimiento de patrones

Juan Francisco Díaz Frias Emily Núñez

Marzo 2025

1. Maniobras en trenes

En este taller se trabajará un problema de maniobra de trenes en una estación de maniobras. **El objetivo es adquirir destreza para escribir código funcional usando la técnica de reconocimiento de patrones.**

Usted está a cargo de realizar maniobras en los vagones de un tren. Para esto se asume que cada vagón se puede maniobrar individualmente y que el tren no tiene un motor específico.

La tarea de maniobrar está especificada por dos secuencias de vagones: el tren dado y el tren deseado. El proyecto consiste en reordenar el tren dado con la ayuda de estaciones auxiliares de forma tal que se obtenga el tren deseado. No se espera solamente que reordene los vagones del tren, sino que también compute una secuencia de movimientos de maniobra (que de ahora en adelante se llamarán solamente "movimientos".)

La estación de maniobras se muestra en la figura 1. Tiene un trayecto "principal" y trayectos para maniobras "uno" y "dos". Una situación en la estación de maniobras se conoce como un *estado*. Un *movimiento* describe cómo los vagones se mueven de un trayecto a otro.

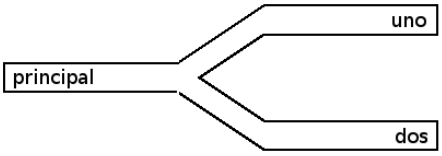


Figura 1: Estación de maniobras.

El objetivo final es encontrar una secuencia de movimientos que conviertan el tren del trayecto "principal", en otra configuración del tren sobre el trayecto "principal".

1.1. El modelo

- Trenes, vagones y estados. Un vagón es un dato de cualquier tipo (por ejemplo enteros, o caracteres, o .....). Un tren es una lista de vagones. Un tren no tiene vagones duplicados (esto es, $List('a','b')$ es un tren, pero $List('a','a')$ no). Una descripción completa del estado de una estación de maniobras consiste en tres listas: una lista que describe el trayecto "principal", y dos listas que describen los trayectos "uno" y "dos". Un estado completo está representado como una tripla que tiene las correspondientes listas de trenes que hay en cada uno de los trayectos de la estación.

  El estado

  $$(List('a','b'), List('d'), List('c'))$$

  se puede puede visualizar en la siguiente figura:

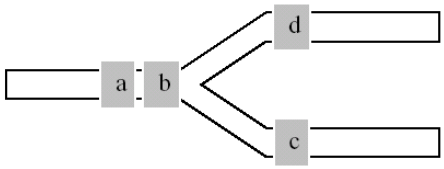


Figura 2: Ejemplo de un estado.

Los tipos de datos que usaremos en este ejercicio, en Scala, son:

```
type Vagon = Any
type Tren = List[Vagon]
type Estado = (Tren, Tren, Tren)
```

- Movimientos. Los movimientos se representan con *clases case*.

```
trait Movimiento
case class Uno(n:Int) extends Movimiento
case class Dos(n:Int) extends Movimiento
```

Hay dos tipos de movimientos: *Uno* y *Dos*. Cada movimiento tiene como parámetro un entero. Ejemplos de movimientos son: $Uno(1)$, $Dos(3)$, $Uno(-2)$.

Aplicando un Movimiento a un Estado. Los movimientos determinan cómo un estado se transforma en otro:

- Si el movimiento es $Uno(n)$ y $n > 0$, entonces los n vagones de más a la derecha, son movidos del trayecto "principal" al trayecto "uno".

  Si hay más de n vagones en el trayecto "principal", los otros vagones se mantienen allí. Si hay menos se trasladan todos sin producirse ningún error.
- Si el movimiento es $Uno(n)$ y $n < 0$, entonces los n vagones de más a la izquierda, son movidos del trayecto "uno" al trayecto "principal".

Si hay más de n vagones en el trayecto "uno", los otros vagones se mantienen allí. Si hay menos se trasladan todos sin producirse ningún error.

- El movimiento $Uno(0)$ no tiene ningún efecto.

De manera análoga funciona para movimientos del tipo *Dos*, concernientes al trayecto "dos".

1.2. Ejercicios de programación

1.2.1. Aplicar movimiento

Implemente en Scala la función `aplicarMovimiento` que recibe un estado e y un movimiento m y devuelve el estado resultante de aplicarle el movimiento m sobre la estación de maniobras con estado e.

```
  def aplicarMovimiento(e:Estado, m:Movimiento): Estado = {
...
  }
```

Por ejemplo:

```
val e1 = (List('a', 'b', 'c', 'd'), Nil, Nil)
val e2= aplicarMovimiento(e1,Uno(2))
val e3 = aplicarMovimiento(e2,Dos(3))
val e4 = aplicarMovimiento(e3, Dos(-1))
val e5 = aplicarMovimiento(e4,Uno(-2))
```

devuelve:

```
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(a),List(c, d),List(b))
val e5: ManiobrasTrenes.Estado = (List(a, c, d),List(),List(b))
```

1.2.2. Aplicar movimientos

Implemente en Scala la función `aplicarMovimientos` que recibe un estado e y una lista de movimientos *movs* y devuelve la lista de estados por las que pasa la estación de maniobras desde que inicia en el estado e hasta que se aplica el último de los movimientos en *movs*.

```
  def aplicarMovimientos(e:Estado, movs:Maniobra): List[Estado] = {
...
  }
```

Así si la lista *movs* tiene n elementos, el resultado es una lista con $n + 1$ elementos.
Por ejemplo:

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

devuelve:

```
val e: (List[Char], List[Char], List[Char]) = (List(a, b),List(c),List(d))
val res1: List[ManiobrasTrenes.Estado] = List((List(a, b),List(c),List(d)),
(List(a),List(b, c),List(d)),
(List(),List(b, c),List(a, d)),
(List(b, c),List(),List(a, d)))
```

En la siguiente figura se muestra cómo evoluciona la estación de maniobras de un estado a otro:

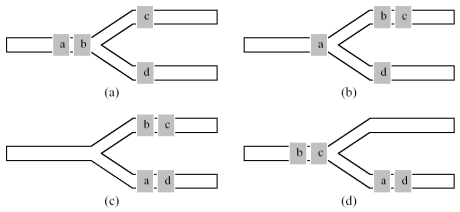


Figura 3: Ejemplo de transición de un estado a otro.

1.2.3. Definir maniobras

Por último, se desea definir la maniobra que se debe hacer para convertir un tren $t1$ sobre el trayecto "principal" en un tren $t2$ sobre el mismo trayecto, suponiendo que los trayectos "uno" y "dos" están desocupados inicialmente. Obviamente al final también estarán desocupados. Por *maniobra* se entenderá la lista de movimientos que se deben realizar para lograr el objetivo.

```
type Maniobra = List[Movimiento]
```

Implemente la función `definirManiobra` que recibe dos trenes $t1$ y $t2$ que devuelve una maniobra *movs* tal que al aplicar los movimientos en el orden en que vienen en la maniobra se logra pasar a la estación del estado $(t1, Nil, Nil)$ al estado $(t2, Nil, Nil)$.

Por ejemplo:

```
definirManiobra(List('a', 'b', 'c', 'd'), List('d', 'b', 'c', 'a'))
```

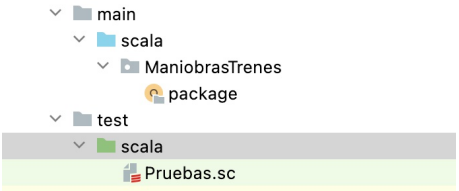
da como resultado:

```
val res2: ManiobrasTrenes.Maniobra = List(Uno(4), Uno(-3), Dos(3), Uno(-1),
                                          Dos(-1), Uno(1), Dos(-1), Dos(-1), Uno(-1))
```

2. Entrega

2.1. Paquete *ManiobraTrenes* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:



Las funciones correspondientes a cada ejercicio, `aplicarMovimiento`, `aplicarMovimientos` y `definirManiobra` deben ser implementadas en un paquete de Scala denominado `ManiobraTrenes`. **Todas deben usar reconocimiento de patrones esencialmente.** En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object ManiobrasTrenes {
  type Vagon = Any
  type Tren = List[Vagon]
  type Estado = (Tren, Tren, Tren)

  trait Movimiento
  case class Uno(n:Int) extends Movimiento
  case class Dos(n:Int) extends Movimiento

  type Maniobra = List[Movimiento]

  def aplicarMovimiento(e:Estado, m:Movimiento): Estado = {
...
  }

  def aplicarMovimientos(e:Estado, movs:Maniobra): List[Estado] = {
...
  }

  def definirManiobra(t1:Tren, t2:Tren): Maniobra = {
    // Dados dos trenes t1 (el original) y t2 (el deseado), devuelve la maniobra que se necesita hacer
    // para pasar una estacion de maniobras del estado (t1, List(), List()) al estado (t2, List(), List())
...
}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import ManiobrasTrenes._

val e1 = (List('a', 'b', 'c', 'd'), Nil, Nil)
val e2= aplicarMovimiento(e1,Uno(2))
val e3 = aplicarMovimiento(e2,Dos(3))
val e4 = aplicarMovimiento(e3, Dos(-1))
val e5 = aplicarMovimiento(e4,Uno(-2))

aplicarMovimientos(e1, List(Uno(2), Dos(3), Dos(-1), Uno(-2), Dos(-1)))
val e = (List('a', 'b'), List('c'), List('d'))
aplicarMovimientos(e, List(Uno(1), Dos(1), Uno(-2)))
definirManiobra(List('a', 'b', 'c', 'd'), List('d', 'b', 'c', 'a'))
```

2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de uso del reconocimiento de patrones, informe de corrección y conclusiones.

2.2.1. Informe de uso del reconocimiento de patrones

Tal como se ha visto en clase, el reconocimiento de patrones es una herramienta my poderosa y expresiva para programar. En esta sección usted debe hacer una tabla indicando en cuáles funciones usó la técnica y en cuáles no. Y en las que no, indicar por qué no lo hizo.

También se espera una apreciación corta de su parte, sobre el uso del reconocimiento de patrones como técnica de programación.

2.2.2. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de las funciones entregadas, y también deberá entregar un conjunto de pruebas. Todo esto lo consigna en esta sección del informe, dividida de la siguiente manera:

Argumentación sobre la corrección Para cada función, `aplicarMovimiento`, `aplicarMovimientos` y `definirManiobra`, argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

Casos de prueba Para cada función se requieren mínimo 5 casos de prueba **propios, diferentes a los del enunciado**, donde se conozca claramente el valor esperado, y se pueda evidenciar que el valor calculado por la función corresponde con el valor esperado en toda ocasión.

Una descripción de los casos de prueba diseñados, sus resultados y una argumentación de si cree o no que los casos de prueba son suficientes para confiar en la corrección de cada uno de sus programas, los registra en esta sección del informe. Obviamente, esta parte del informe debe ser coherente con el archivo de pruebas entregado, `pruebas.sc`.

2.3. Fecha y medio de entrega

Todo lo anterior, es decir los archivos `package.scala`, `pruebas.sc`, e `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 3 de abril de 2025, en un archivo comprimido que traiga estos tres elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación funcional*.

3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.
- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.