Taller 3

Reconocimiento de patrones

Estudiantes:

Heidy Mina - 201931720
Juan José Bolaños - 201942124

Asignatura: Fundamentos de Programación Funcional y Concurrente

Docente: Juan Francisco Díaz Frias

Universidad del Valle

Santiago de Cali

Septiembre - 2023

Contenido

El modelo

- Primero se crearon tres tipos de datos, uno llamado Vagón, el cual puede ser de cualquier tipo, otro llamado Tren, el cual es una lista de elementos de tipo Vagón y otro llamado Estado, el cual es un vector de elementos de tipo Tren.

```
1. type Vagon = Any
2. type Tren = List[Vagon]
3. type Estado = (Tren, Tren, Tren)
```

- Luego se creó un trait llamado Movimiento para utilizar en los dos tipos de movimientos que utilizaremos: Uno y Dos. Los movimientos se representan con clases case. Este trait contiene una función llamada 'cambiarEstado', la cual recibe un elemento de tipo Estado y devuelve otro del mismo tipo.

```
1. trait Movimiento {
2.   def cambiarEstado(estadoActual: Estado) : Estado
3. }
```

Aplicando un Movimiento a un Estado

Movimiento Uno

- 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 vagones en el trayecto "principal", los otros vagones se mantienen allí. Si hay menos se trasladan todos sin producir 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 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.

```
1. case class Uno(n: Int) extends Movimiento {
2.   def cambiarEstado(estadoActual: Estado): Estado = estadoActual match {
3.     case (principal, uno, dos) if n > 0 =>
4.       val(vagonesSinMover,vagonesAMover) = if (n > principal.length){
5.         (Nil, principal)
6.       }
7.       else principal.splitAt(n)
8.       (vagonesSinMover, vagonesAMover ++ uno , dos)
9.     case (principal, uno, dos) if n < 0 =>
```

```
      val(vagonesAMover,vagonesSinMover) = uno.splitAt(-n)
      (principal ++ vagonesAMover, vagonesSinMover, dos)
    case (principal, uno, dos) if n == 0 => estadoActual
  }
}
```

Movimiento Dos

- Para el movimiento Dos se utilizaron los mismos casos y condiciones que se utilizaron en el movimiento Uno.

```
case class Dos(n: Int) extends Movimiento {
  def cambiarEstado(estadoActual: Estado): Estado = estadoActual match {
    case (principal, uno, dos) if n > 0 =>
      val(vagonesSinMover,vagonesAMover) = if (n >= principal.length){
        (Nil, principal)
      }
      else principal.splitAt(n)
      (vagonesSinMover, uno , vagonesAMover ++ dos)
    case (principal, uno, dos) if n < 0 =>
      val(vagonesAMover,vagonesSinMover) = dos.splitAt(-n)
      (principal ++ vagonesAMover, uno , vagonesSinMover)
    case (principal, uno, dos) if n == 0 => estadoActual
  }
}
```

Ejercicios de programación

Aplicar movimiento

Se creó una función llamada ‘aplicarMovimiento’ que recibe como parámetros un estado y un movimiento. Lo que contiene son dos casos, uno llamado u, que es para el Movimiento tipo Uno y otro llamado d, que es para el movimiento tipo Dos. Cada uno de estos casos acceden a la función ‘cambiarEstado’ de su respectiva clase con el Estado recibido como parámetro evaluado.

```
def aplicarMovimiento(e: Estado, m: Movimiento): Estado = m match {
  case u: Uno => u.cambiarEstado(e)
  case d: Dos => d.cambiarEstado(e)
}
```

Aplicar movimientos

Se creó una función llamada ‘aplicarMovimientos’ que recibe como parámetros un estado y una lista de movimientos. También se creó otro tipo de dato llamado Maniobra, que es una lista de Movimientos. Esta función contiene dos casos, uno llamado no_terminado, el cual verifica que la lista de movimientos no esté vacía para posteriormente crear una lista que tiene como primer elemento una variable ‘first’ que contiene la función ‘aplicarMovimiento’ evaluada con el estado y el movimiento ubicado en la posición 0 de la lista de movimientos, este primer elemento se concatena con la lista que devuelve la función ‘aplicarMovimientos’ evaluada con la variable first y el último elemento de la lista de movimientos de entrada. El otro caso es el caso m, el cual verifica que la lista de movimientos esté vacía, si lo está, devuelve una lista de Estados vacía.

```
1. def aplicarMovimientos(e: Estado, movs: Maniobra):
   List[Estado] = movs match {
2.   case no_terminado: Maniobra if !movs.isEmpty =>
3.     val first = aplicarMovimiento(e, movs(0))
4.     first :: aplicarMovimientos(first, movs.tail)
5.   case m: Maniobra if movs.isEmpty => List[Estado]()
6. }
```

Definir maniobras

La función definirManiobra está diseñada para realizar una serie de operaciones sobre dos trenes representados como listas (Tren) y determinar una secuencia de maniobras necesarias para que el tren t1 coincida con el tren t2, teniendo en cuenta ciertas restricciones.
Parámetros de entrada:

t1: Tren - Representa el primer tren, que es una lista de elementos.
t2: Tren - Representa el segundo tren, que también es una lista de elementos.

Descripción de la función:
La función definirManiobra se encarga de calcular una secuencia de maniobras necesarias para transformar el tren t1 en el tren t2 teniendo en cuenta ciertas condiciones. Para realizar este cálculo, se llama a la función auxiliar definirManiobra_recursiva con valores iniciales true y true para los parámetros uno y dos, respectivamente.

definirManiobra_recursiva

Parámetros de entrada:

t1: Tren - El tren actual, que es una lista de elementos.
t2: Tren - El tren objetivo, que también es una lista de elementos.
uno: Boolean - Un valor booleano que indica si se permite un tipo de maniobra.
dos: Boolean - Un valor booleano que indica si se permite otro tipo de maniobra.

Descripción de la función auxiliar:

Se calcula el tamaño (tamano) del tren actual t1.
Se utiliza el patrón match para evaluar varios casos posibles:
Si t1 y t2 son idénticos, se devuelve una lista con un movimiento Uno(0).
Si el primer vagón de t1 es igual al primer vagón de t2, se llama recursivamente a la función con t1.tail y t2.tail.
Si el primer vagón de t1 es diferente al primer vagón de t2, y se permite un tipo de maniobra, se genera una secuencia de maniobras.
Si el último vagón de t1 es igual al primer vagón de t2, y se permite otro tipo de maniobra, se genera una secuencia de maniobras.
Si el primer vagón de t1 es igual al último vagón de t2, y se permiten ambas maniobras, se genera una secuencia de maniobras.
Si t1 y t2 son idénticos y se permiten ambas maniobras, se genera una secuencia especial.
Si ninguna de las condiciones anteriores se cumple y t1 está vacío (caso base), se devuelve una lista vacía de movimientos.

La función recursiva se llama con diferentes combinaciones de valores uno y dos, y se concatenan las secuencias de maniobras según las condiciones definidas. Cada maniobra se representa como un objeto de tipo Movimiento, que puede ser de tipo Uno o Dos, con valores numéricos que indican el movimiento a realizar.

Reconocimiento de patrones

Función	**Reconocimiento de patrones**

‘cambiarEstado(estadoActual : Estado) : Estado’	No hace falta usar reconocimiento de patrones en la función base.
‘case class Uno(n: Int) extends Movimiento’	Se crea una ‘case class’ para el Movimiento Uno, las cuales contienen varios casos de la función ‘cambiarEstado’.
‘case class Dos(n: Int) extends Movimiento’	Se crea una ‘case class’ para el Movimiento Dos, las cuales contienen varios casos de la función ‘cambiarEstado’.
‘aplicarMovimiento(e: Estado, m: Movimiento): Estado = m match’	Se utiliza la expresión match, la cual es usada para comparar el parámetro Movimiento con cualquiera de los ya existentes tipos de movimientos, en este caso representados con los casos u y d.
‘aplicarMovimientos(e: Estado, movs: Maniobra): List[Estado] = movs match’	Se utiliza la expresión match, la cual es usada para comparar el parámetro Maniobra con los casos no_terminado y m.
efinirManiobra_recursiva(t1: Tren, t2: Tren, uno: Boolean, dos: Boolean): Maniobra = {	se usa la expresión match para comparar a t1 con t2

Casos de prueba

Se hará la prueba con cada función evaluando varios Estados y Movimientos, comentando el resultado esperado

Aplicar movimiento

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

Aplicar movimientos

```
1. val e6 = (List('a', 'b'), List('c'), List('d')) //(List(a,
   b),List(c),List(d))
2. val e7 = aplicarMovimientos(e6, List(Uno(1), Dos(1), Uno(-3))) //
   List((List(a),List(b, c),List(d)), (List(),List(b, c),List(a, d)), (List(b,
   c),List(),List(a, d)))
3. val e8 = aplicarMovimientos(e6, List(Dos(1), Uno(1), Uno(-2))) //
   List((List(a),List(c),List(b, d)), (List(a),List(c),List(b, d)), (List(a,
   c),List(),List(b, d)))
4. val e9 = aplicarMovimientos(e6, List(Uno(-1), Uno(1), Uno(-1))) //
   List((List(a, b, c),List(),List(d)), (List(a),List(b, c),List(d)), (List(a,
   b),List(c),List(d)))
5. val e10 =aplicarMovimientos(e6, List(Dos(3), Uno(-1), Uno(-2))) //
   List((List(),List(c),List(a, b, d)), (List(c),List(),List(a, b, d)),
   (List(c),List(),List(a, b, d)))
```