

Taller 1: Funciones y procesos

Fundamentos de Programación Funcional y Concurrente

Juan José Millán Hernández – 2266393

Andrés Felipe Chaparro Pulgarin – 2266252

Jose David Marmol Otero – 2266370

Ingeniería de sistemas

Carlos Andrés Delgado S

Universidad del valle – Tuluá

16 de septiembre del 2024

1. Informe de proceso

Descripción de ejercicios: El primer ejercicio consiste en calcular el valor máximo de una lista de enteros positivos no vacía. Se implementan dos funciones: `maxLin` (recursión lineal) y `maxIt` (recursión de cola).

El segundo ejercicio se basa en el problema de las Torres de Hanoi. En este problema, tienes que mover discos entre tres torres, siguiendo la regla de que no puedes poner un disco más grande sobre uno más pequeño. Para resolverlo, se usan dos funciones: `movsTorresHanoi` calcula el número mínimo de movimientos necesarios, y `torresHanoi` genera la lista detallada de todos los movimientos necesarios para trasladar los discos de la torre inicial a la torre final.

1. Máximo de una Lista de Enteros

Descripción de la Función:

- ✚ **maxLin:** Esta función utiliza recursión lineal. Compara el primer elemento de la lista con el máximo del resto de la lista, y utiliza `math.max` para conservar el mayor valor. La recursión finaliza cuando se llega al último elemento de la lista.
- ✚ **maxIt:** Esta función emplea recursión de cola para optimizar el uso de la pila de llamadas. Utiliza una función auxiliar `maxItAux` que actualiza el valor máximo en cada paso, comparando el valor actual de la lista con el máximo acumulado.

Ejemplos y Proceso de Ejecución:

Ejemplo maxLin:

Código:

```
def maxLin(l: List[Int]): Int = {  
  l match {  
    case Nil => throw new NoSuchElementException("Lista vacía")  
    case head :: Nil => head  
    case head :: tail => math.max(head, maxLin(tail))  
  }  
}
```

- Ejemplo de Ejecución:
- Pila de llamadas:

Ejemplo maxIt:

Código:

```


•   def maxIt(l: List[Int]): Int = {
•       @tailrec
•       def maxItAux(l: List[Int], max: Int): Int = {
•           l match {
•               case Nil => max
•               case head :: tail => maxItAux(tail, math.max(max, head))
•           }
•       }
•
•       l match {
•           case Nil => throw new NoSuchElementException("Lista vacía")
•           case head :: tail => maxItAux(tail, head)
•       }
•   }
• }

```

- Ejemplo de Ejecución:
- Pila de llamadas:

1. Torres de Hanoi

Descripción de la Función:

 **movsTorresHanoi:** Para calcular el número mínimo de movimientos necesarios para resolver un problema de Torres de Hanoi con un numero n de discos podemos utilizar la siguiente formula: $B(n) = 2^n - 1$

En un caso general para n discos, podemos dividir el problema en 3 partes:


*Mover los $n - 1$ discos superiores a una torre auxiliar.

*Mover el disco más grande a la torre final.

*Mover los $n - 1$ discos desde la torre auxiliar a la torre fina la cual necesita al menos $B(n - 1)$ movimientos.

Por lo tanto, podríamos ver el total de movimientos como: $B(n) = 2 * B(n - 1) + 1$.

De otra forma obtenemos: $B(n) = 2^n - 1$

 **torresHanoi:** Para generar una lista de movimientos necesarios para trasladar los discos entre las Torres de Hanoi, se utiliza un enfoque recursivo.

Para mover n discos de la Torre inicial a la de destino, se comienza moviendo los $n - 1$ discos más pequeños desde la Torre inicial a la Torre auxiliar, Este paso se realiza usando la torre destino como una torre intermediaria. Una vez que estos discos están en la torre auxiliar, se mueve el disco más grande, el disco n , directo

desde la torre inicial a la de destino. Después de trasladar el disco más grande, se mueven los restantes de la torre auxiliar a la de destino. Este proceso se repite de forma recursiva para cada conjunto de discos, generando una lista de movimientos que describe cómo trasladar todos los discos de la torre inicial a la torre destino de forma correcta.

Ejemplos y Proceso de Ejecución:

Ejemplo `movsTorresHanoi`:

- **Código de Ejemplo:**
- **Ejemplo de Ejecución:**

Ejemplo `torresHanoi`:

- **Código de Ejemplo:**
- **Ejemplo de Ejecución:**
- **Pila de llamadas:**

2. Informe de Corrección

1. Argumentación sobre la Corrección

`maxLin` y `maxIt`:

- ✚ **`maxLin`:** La función `maxLin` encuentra el valor máximo en una lista usando recursión. Compara el primer elemento con el máximo del resto de la lista. Esto asegura que al final, el valor máximo de toda la lista se haya considerado. Si la lista tiene un solo elemento, ese es el máximo.
- ✚ **`maxIt`:** La función `maxIt` usa recursión de cola, lo que hace que sea más eficiente al evitar que se acumule información en la pila. Compara el valor actual con el máximo encontrado hasta ese momento y actualiza el máximo si es necesario. Esto evita problemas de desbordamiento de pila y mejora el rendimiento, especialmente con listas largas.

`movsTorresHanoi` y `torresHanoi`:

- ✚ La función `movsTorresHanoi` tiene el objetivo calcular el mínimo de movimientos necesarios para trasladar n discos desde una torre de inicio o de origen a una torre de destino utilizando una torre auxiliar. La clave para entender esta función es la fórmula matemática fundamental del problema de las Torres de Hanoi, que

establece que el número mínimo de movimientos necesarios para trasladar n discos es $2^n - 1$.

La corrección de la formula se basa en que para resolver el problema con n discos, primero se deben mover $n - 1$ discos a una torre intermedia o auxiliar para luego moverla a la torre de destino y por último trasladar los $n - 1$ discos restantes de la torre auxiliar al destino.

La función `torresHanoi` es la encargada de generar una lista de movimientos necesarios para trasladar una cantidad n de discos de una torre inicial a su destino mediante una torre auxiliar. Para encontrar la solución de este problema podemos basarnos en un enfoque recursivo que divide el problema principal en subproblemas.

- ✚ Primeramente, de nuevo la función mueve $n - 1$ discos desde la torre inicial a la auxiliar, usando la de torre final como intermedia. Luego, mueve el disco que es más grande directamente desde la torre de origen a la torre de destino. Por último, mueve los $n - 1$ discos desde la torre auxiliar a la torre destino. Este proceso recursivo permite asegurarnos que todos los discos se trasladan siguiendo las reglas

2. Casos de Prueba

maxLin:

```
// MaximoListaTest.scala
import org.scalatest.funSuite.AnyFunSuite
import taller1.MaximoLista

class MaximoListaTest extends AnyFunSuite {

  val maximo = new MaximoLista

  // Casos de prueba para maxLin
  test("maxLin debe devolver el valor máximo en una lista de enteros positivos") {
    assert(maximo.maxLin(List(3, 5, 2, 8, 1)) == 8)
  }

  test("maxLin debe lanzar NoSuchElementException para una lista vacía") {
    assertThrows[NoSuchElementException] {
      maximo.maxLin(List())
    }
  }

  test("maxLin debe devolver el valor del único elemento para una lista con solo un elemento") {
    assert(maximo.maxLin(List(7)) == 7)
  }
}
```

```

    }

    test("maxLin debe manejar una lista con enteros negativos y positivos") {
        assert(maximo.maxLin(List(-3, 5, -2, 8, 1)) == 8)
    }

    test("maxLin debe manejar una lista donde todos los elementos son iguales") {
        assert(maximo.maxLin(List(4, 4, 4, 4)) == 4)
    }
}

```

MaxIt:

```

// Casos de prueba para maxIt
test("maxIt debe devolver el valor máximo en una lista de enteros positivos") {
    assert(maximo.maxIt(List(3, 5, 2, 8, 1)) == 8)
}

test("maxIt debe lanzar NoSuchElementException para una lista vacía") {
    assertThrows[NoSuchElementException] {
        maximo.maxIt(List())
    }
}

test("maxIt debe devolver el valor del único elemento para una lista con un solo elemento") {
    assert(maximo.maxIt(List(7)) == 7)
}

test("maxIt debe manejar una lista con enteros negativos y positivos") {
    assert(maximo.maxIt(List(-3, 5, -2, 8, 1)) == 8)
}

test("maxIt debe manejar una lista donde todos los elementos son iguales") {
    assert(maximo.maxIt(List(4, 4, 4, 4)) == 4)
}
}

```

movsTorresHanoi:

torresHanoi:

3. Conclusiones

Reflexión sobre el Proceso: Programar con pequeñas listas en maxLin y maxIt nos ayudó a entender mejor cómo funciona la recursión lineal y de cola, ver las recursiones manejar la pila de llamadas y como se resuelven los problemas fue útil. Sin embargo, fue difícil que maxIt manejase bien el valor máximo en las llamadas, solo tras muchas pruebas se logró implementar. Con todo esto nos queda como conclusión el aprendizaje de la utilidad y manejo de las recursiones lineal y de cola, son funciones que si se escriben de manera optimizada pueden ser de mucha utilidad.