UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y Ciencias de la Computación

Programación avanzada II

Lab 1. Un primer contacto con Scala

1. Escribe una función recursiva de cola `primeFactors(n: Int): List[Int]` que devuelva una lista con los factores primos de un entero positivo dado n. Ejemplos:

```
println(primeFactors(60))   // Output: List(2, 2, 3, 5)
println(primeFactors(97))   // Output: List(97)
println(primeFactors(84))   // Output: List(2, 2, 3, 7)
```

Solución

```
def primeFactors(n: Int): List[Int] =
  @scala.annotation.tailrec
  def factorize(n: Int, divisor: Int, acc: List[Int]): List[Int] =
    if (n < 2)
      acc
    else if (n % divisor == 0)
      factorize(n / divisor, divisor, acc :+ divisor)
    else factorize(n, divisor + 1, acc)

  factorize(n, 2, List())
```

2. Escribe una función recursiva de cola `binarySearch(arr: Array[Int], elt: Int): Option[Int]` que devuelva el índice de elt (`Some(i)`) en un array ordenado utilizando el algoritmo de búsqueda binaria, o `None` en caso de que el elemento no se esté. Ejemplos:

```
val arr = Array(1, 3, 5, 7, 9, 11)
println(binarySearch(arr, 5))  // Output: Some(2)
println(binarySearch(arr, 10)) // Output: None
```

Solución

```
def binarySearch(arr: Array[Int], target: Int): Option[Int] =
  @scala.annotation.tailrec
  def search(low: Int, high: Int): Option[Int] =
    if (low > high)
      None // Base case: Target not found
    else {
      val mid = low + (high - low) / 2
      if (arr(mid) == target)
        Some(mid) // Found the target
      else if (arr(mid) > target)
        search(low, mid - 1) // Search in the left half
      else
        search(mid + 1, high) // Search in the right half
    }

  search(0, arr.length - 1)
```

3. Define una función recursiva genérica `unzip` que tome una lista de tuplas con dos componentes y que devuelva una tupla con dos listas: una con las primeras componentes y otra con las segundas. Por ejemplo,

```
unzip(List((10, 'a'), (20, 'b'), (30, 'c'))
== (List(10, 20, 30), List('a', 'b', 'c'))
```

Solución 1 (Pattern matching)

```
def unzip[A,B](l:List[(A,B)]): (List[A], List[B]) =
  l match
    case Nil => (Nil, Nil)
    case (a, b) :: t =>
      val (as, bs) = unzip(t)
      (a :: as, b :: bs)
```

Solución 2 (Iterativa)

```
def unzip[A,B](l: List[(A,B)]): (List[A], List[B]) =
  var l0 = List[A]()
  var l1 = List[B]()
  for (e <- l)
    l0 = l0 :+ e(0)
    l1 = l1 :+e(1)
  (l0, l1)
```

Solución 3 (Recursiva de cola)

```
def unzip[A,B](l: List[(A,B)]) : (List[A],List[B]) =
  @annotation.tailrec
  def go(l: List[(A,B)], la: List[A], lb: List[B]) : (List[A], List[B]) =
    if l.isEmpty then (la,lb)
    else go(l.tail, la :+ l.head(0), lb :+ l.head(1))

  go(l, Nil, Nil)
```

4. Define una función recursiva genérica `zip` que tome dos listas y devuelva una lista de tuplas, donde las primeras componentes se tomen de la primera lista y las segundas componentes de la segunda lista. Por ejemplo:

```
zip(List(10, 20, 30), List('a', 'b', 'c'))
== List((10, 'a'), (20, 'b'), (10, 'c'))
zip(List(10, 20, 30), List('a', 'b'))
== List((10,'a'), (20,'b'))
```

Solución 1 (Pattern Matching)

```
def zip[A,B](l1:List[A], l2:List[B]): List[(A,B)] =
  (l1, l2) match
    case (Nil, _) => Nil
    case (_, Nil) => Nil
    case (h1 :: t1, h2 :: t2) => (h1, h2) :: zip(t1, t2)
```

Solución 2 (Recursiva de cola)

```
def zip[A,B](la: List[A], lb: List[B] ) : List[(A,B)] =
  @annotation.tailrec
  def go(la: List[A], lb: List[B], lc: List[(A,B)]) : List[(A,B)] =
    if la.isEmpty || lb.isEmpty then lc
    else go(la.tail, lb.tail, lc :+ (la.head,lb.head))

  go(la, lb, Nil)
```

5. Implementa una operación filter(l, f) que tome una lista l de elementos de tipo A y una función f: A => Boolean y que devuelva una lista con los elementos e de l que satisfacen f(e). Por ejemplo:

```
println(filter(List(1,2,3,4,5), _ % 2 == 0)) // Output: List(2,4)
```

Solución 1 (Pattern Matching)

```
def filter[A](l:List[A], f: A => Boolean): List[A] =
  l match
    case Nil => Nil
    case h :: t =>
      if f(h) then h :: filter(t, f)
      else filter(t, f)
```

Solución 2 (Recursiva de cola)

```
def filter[A](l: List[A], f: A => Boolean) : List[A] =
  @annotation.tailrec
  def loop(l: List[A], f: A => Boolean, lres : List[A]) : List[A] =
    if l.isEmpty then lres
    else if f(l.head) then loop(l.tail,f,lres :+ l.head)
    else loop(l.tail,f,lres)

  loop(l,f,Nil)
```

6. Implementa una operación map(l, f) que tome como argumentos una lista l de elementos de tipo A y una función f: A => B y que devuelva una lista de elementos de tipo B con los elementos resultantes de aplicar f a cada uno de los elementos de l.

```
println(map(List(1,2,3,4,5), _ * 2)) // Output: List(2,4,6,8,10)
```

Solución 1 (Pattern Matching)

```
def map[A,B](l:List[A], f: A => B): List[B] =
  l match
    case Nil => Nil
    case h :: t => f(h) :: map(t, f)
```

Solución 2 (Recursiva de cola)

```
def map[A,B](l: List[A], f : A => B)  : List[B] =
  @annotation.tailrec
  def loop(l: List[A], f: A => B, lb: List[B]): List[B] =
    if l.isEmpty then lb
    else loop(l.tail, f, lb :+ f(l.head))

  loop(l, f, Nil)
```

7. Implementa una operación groupBy(l, f) que tome como argumentos una lista l de elementos de tipo A y una función f: A => B y que devuelva un objeto de tipo Map[B, List[A]] que asocie una lista con los elementos e de l con el mismo f(e).

```
println(groupBy(List(1,2,3,4,5), _ % 2 == 0))
      // Output: Map(false -> List(5, 3, 1), true -> List(4, 2))
```

Solución 1 (Iterativa – Map mutable)

```
def groupBy[A,B](l:List[A], f: A => B): Map[B, List[A]] =
  val map = collection.mutable.Map[B, List[A]]()
  for e <- l do
    val key = f(e)
    if map.contains(key) then
      map(key) = e :: map(key)
    else
      map(key) = List(e)
  map.toMap
```

Solución 2 (Recursiva de cola – Map inmutable)

```
def groupBy[A,B](l: List[A], f: A => B): Map[B, List[A]] =
  @annotation.tailrec
  def loop(l: List[A], f: A => B, m: Map[B, List[A]]): Map[B, List[A]] =
    if l.isEmpty then m
    else loop(l.tail, f, m.updated(f(l.head), m.getOrElse(f(l.head), Nil)
:+ l.head))

  loop(l, f, Map.empty)
```

8. Implementa una operación reduce(l, f) que toma como argumentos una lista l de elementos de tipo A y una función f de tipo (A, A) => A y que devuelva el resultado de combinar todos los elementos de l utilizando la función f. Por ejemplo:

```
println(reduce(List(1,2,3,4,5), _ + _)) // Output: 15
```

Solución 1 (Iterativa)

```
def reduce[A](l:List[A], f: (A, A) => A): A =
  require(l.nonEmpty)
  var acc = l.head
  for e <- l.tail do
    acc = f(acc, e)
  acc
```

Solución 2 (Recursiva de cola)

```
def reduce[A](l: List[A], f : (A,A) => A) : A =
  @annotation.tailrec
  def loop(l : List[A], f: (A,A) => A, res: A) : A =
    if l.isEmpty then res
    else loop(l.tail, f, f(res,l.head))

  loop(l.tail, f, l.head)
```

9. Implementa una función recursiva para generar todos los subconjuntos de un conjunto determinado. Conviértela en recursiva de cola.

```
println(subsets(Set())) // Output: Set(Set())
println(subsets(Set(1))) // Output: Set(Set(), Set(1))
println(subsets(Set(1,2))) // Output: Set(Set(),Set(1),Set(2),Set(1,2))
println(subsets(Set(1, 2, 3)))
// Output: Set(Set(),Set(1),Set(2),Set(1,2),Set(3),Set(1,3),Set(2,3),Set(1,2,3))
```

Solución 1 (Iterativa)

```
def subsets[A](set: Set[A]): Set[Set[A]] = {
  if (set.isEmpty) Set(Set())
  else {
    val rest = subsets(set.tail)
    rest ++ rest.map(_ + set.head)
  }
}
```

Solución 2 (Recursiva de cola)

```
def subsets[A](set: Set[A]): Set[Set[A]] = {
  @scala.annotation.tailrec
  def subsetsAux(set: Set[A], acc: Set[Set[A]]): Set[Set[A]] = {
    if (set.isEmpty) acc
    else {
      val rest = acc.map(_ + set.head)
      subsetsAux(set.tail, acc ++ rest)
    }
  }

  subsetsAux(set, Set(Set()))
}
```

10. Escribe una función recursiva de cola `generateParentheses(n: Int): List[String]` que genere todas las combinaciones válidas de n pares de paréntesis. Ejemplos:

```
println(generateParentheses(3))
// Output: Lista("((()))", "(()())", "(())()", "()(())", "()()()")
```

Consejos:

- Utiliza un acumulador para almacenar secuencias válidas.
- Haz un seguimiento del número de paréntesis de apertura (open) y cierre (closed) utilizados.
- Caso base: Cuando open == closed == n, agrega la secuencia al resultado.

Solución (Recursiva de cola)

```
def generateParentheses(n: Int): List[String] = {
  @scala.annotation.tailrec
  def generate(open: Int, close: Int, acc: List[String], stack:
List[(String, Int, Int)]): List[String] = {
    stack match {
      case Nil => acc // Base case: no more states to process
      case (current, openLeft, closeLeft) :: rest =>
        if (openLeft == 0 && closeLeft == 0)
          generate(open, close, current :: acc, rest) // Valid sequence
found, add to accumulator
        else {
          val newStack =
            (if (openLeft > 0) (current + "(", openLeft - 1, closeLeft)
:: rest else rest) :::
              (if (closeLeft > openLeft) (current + ")", openLeft,
closeLeft - 1) :: rest else rest)
          generate(open, close, acc, newStack)
        }
    }
  }

  generate(n, n, List(), List(("", n, n)))
}
```