

Fundamentos de Programación Funcional y Concurrente

Listas

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)
juanfco.diaz@correounivalle.edu.co
Edif. B13 - 4009



Universidad del Valle

Septiembre 2023

Plan

1 La descomposición en el diseño de datos

- Interpretador de expresiones aritméticas
- Reconocimiento de patrones
- Listas y Funciones sobre listas

2 Pares y tuplas

- Ordenamiento por mezclas
- Pares y tuplas
- Parametrización de tipos

3 Funciones de alto orden sobre listas

- Map
- Filter
- Reducción de listas
- Reverse

Plan

1 La descomposición en el diseño de datos

- Interpretador de expresiones aritméticas
- Reconocimiento de patrones
- Listas y Funciones sobre listas

2 Pares y tuplas

- Ordenamiento por mezclas
- Pares y tuplas
- Parametrización de tipos

3 Funciones de alto orden sobre listas

- Map
- Filter
- Reducción de listas
- Reverse

Plan

1 La descomposición en el diseño de datos

- Interpretador de expresiones aritméticas
- Reconocimiento de patrones
- Listas y Funciones sobre listas

2 Pares y tuplas

- Ordenamiento por mezclas
- Pares y tuplas
- Parametrización de tipos

3 Funciones de alto orden sobre listas

- Map
- Filter
- Reducción de listas
- Reverse

Contenido

1

La descomposición en el diseño de datos

- Interpretador de expresiones aritméticas
- Reconocimiento de patrones
- Listas y Funciones sobre listas

2

Pares y tuplas

- Ordenamiento por mezclas
- Pares y tuplas
- Parametrización de tipos

3

Funciones de alto orden sobre listas

- Map
- Filter
- Reducción de listas
- Reverse

Interpretador de expresiones aritméticas

- Se quiere escribir un interpretador de expresiones aritméticas.
- Para hacerlo sencillo, se restringirá inicialmente sólo a números y sumas.
- Las expresiones se pueden representar con una jerarquía de clases: una clase (trait) base *Expr* y dos subclases *Numero* y *Suma*.
- Para manipular una expresión se hace necesario conocer su forma y sus componentes.

Expresiones

- La clase *Expr*:

```
0 trait Expr {  
1   def esNumero: Boolean  
2   def esSuma: Boolean  
3   def valorNum: Int  
4   def opIzq: Expr  
5   def opDer: Expr  
6 }
```

- Las clases *Suma* y *Numero*:

```
0 class Numero(n:Int) extends Expr {  
1   def esNumero: Boolean = true  
2   def esSuma: Boolean = false  
3   def valorNum: Int = n  
4   def opIzq: Expr = throw new Error("Numero.opIzq")  
5   def opDer: Expr = throw new Error("Numero.opDer")  
6 }  
7 class Suma(e1:Expr, e2:Expr) extends Expr {  
8   def esNumero: Boolean = false  
9   def esSuma: Boolean = true  
10  def valorNum: Int = throw new Error("Suma.valorNum")  
11  def opIzq: Expr = e1  
12  def opDer: Expr = e2  
13 }
```

Evaluación de Expresiones

- Ahora se quiere escribir una función que evalúe expresiones:

```
0 def eval(e:Expr):Int {  
1     if (e.esNumero) e.valorNum  
2     else if (e.esSuma) eval(e.opIzq) + eval(e.opDer)  
3         else throw new Error("Expresion-desconocida-" + e)  
4 }
```

- Problema:

Escribir todas estas funciones de clasificación y de acceso, se vuelve rápidamente tedioso. ¿Cuántos métodos nuevos se necesitan para incluir, por ejemplo, dos nuevas expresiones *Prod* y *Var*?

Descomposición funcional usando reconocimiento de patrones

- La única razón por la que se necesitan tantas funciones de clasificación y de acceso, es poder **reversar el proceso de construcción** de una expresión:
 - ¿Qué subclase se usó en el proceso?
 - ¿Cuáles fueron los argumentos usados por el constructor en el proceso?
- Esta situación es tan común, que muchos lenguajes de programación la automatizan: **reconocimiento de patrones**
- Una **clase case** es similar a una clase normal, salvo que viene precedida por la palabra **case**:

```
0 trait Expr
1 case class Numero (n:Int) extends Expr
2 case class Suma(e1:Expr, e2:Expr) extends Expr
```

Igual que antes, se define un *trait Expr* y dos subclases *Numero* y *Suma*.

Clases case

- Las clases case para este caso son:

```
0 trait Expr
1 case class Numero (n:Int) extends Expr
2 case class Suma(e1:Expr, e2:Expr) extends Expr
```

- Esta declaración, define implícitamente dos objetos complementarios con métodos *apply*:

```
0 object Numero {
1   def apply(n:Int) = new Numero(n)
2 }
3 object Suma {
4   def apply(e1:Expr, e2:Expr) = new Suma(e1,e2)
5 }
```

de tal forma que se puede escribir *Numero(1)* en lugar de *newNumero(1)*.

- Pero estas clases están vacías. *¿Cómo acceder ahora a sus miembros?*

Reconocimiento de patrones

- El reconocimiento de patrones es una generalización del *Switch* de C/Java, para jerarquías de clases.
En Scala, se usa a través de la expresión *match*:

```
0 def eval(e: Expr): Int = e match {  
1   case Número(n) => n  
2   case Suma(e1, e2) => eval(e1) + eval(e2)  
3   case Prod(e1, e2) => eval(e1) * eval(e2)  
4 }
```

- match* va precedido por una expresión *e* y seguido por una secuencia de **casos**, sintácticamente escritos de la forma
case < pat >=>< expr >.
- Cada caso asocia un patrón *< pat >* con una expresión *< expr >*.
Intuitivamente, si el resultado de evaluar *e* cumple el patrón *< pat >*, se devuelve el resultado de evaluar *< expr >*.
- Si el resultado de evaluar *e* no cumple con ningún patrón, se lanzará una excepción *MatchError*.

Formas de los patrones

- Los patrones se construyen a partir de:
 - Constructores de objetos, e.g. *Numero*, *Suma*, ...
 - Variables, e.g. *e*, *n*, *e₁*, *e₂*, ...
 - Comodines, e.g. -
 - Constantes, e.g. 1, *true*
- Las variables, en los patrones, siempre empiezan por una letra minúscula.
- El mismo nombre de variable no puede aparecer dos o más veces en un mismo patrón. Por ejemplo, *Suma(x, x)* no es un patrón admitido.
- Los nombres de constantes deben empezar con letra mayúscula, a excepción de las palabras reservadas *true*, *false*, *null*

Evaluación de expresiones *match*

- Una expresión de la forma:

$$e \text{ match } \{ \text{case } p_1 \Rightarrow e_1 \dots \text{case } p_n \Rightarrow e_n \}$$

busca hacer corresponder el valor de la evaluación de e con alguno de los patrones p_1, \dots, p_n en el orden en que aparecen.

- Digamos que p_k es el primer patrón que corresponde con la evaluación de e . La expresión completa se substituye por e_k , **haciendo las sustituciones a que haya lugar a causa de esa correspondencia**.
- Específicamente, las referencias a variables del patrón dentro de e_k se substituyen por los valores con que se logró la correspondencia.

¿Cómo se corresponden los patrones y los valores?

- Un patrón construido a partir de un constructor: $C(p_1, \dots, p_n)$ se corresponde con los valores del tipo C (o de un subtipo) que haya sido construido con unos argumentos que se corresponden, cada uno, con los patrones p_1, \dots, p_n .
- Un patrón construido con una variable x se corresponde con cualquier valor, y tiene como efecto **ligar** la variable x con ese valor.
- Un patrón construido con un comodín x se corresponde con cualquier valor (igual al caso de variables, pero no se produce ligadura alguna).
- Un patrón construido con una constante C se corresponde con cualquier valor que sea igual (en el sentido $==$) a esa constante.

Ejemplo de evaluación de correspondencia de patrones

Recordemos *eval*:

```
0 def eval(e:Expr):Int = e match {
1   case Numero(n) => n
2   case Suma(e1,e2) => eval(e1) + eval(e2)
3   case Prod(e1,e2) => eval(e1) * eval(e2)
4 }
```

```
eval(Suma(Numero(1), Numero(2)))
→ Suma(Numero(1), Numero(2)) match {
  case Numero(n) => n
  case Suma(e1, e2) => eval(e1) + eval(e2)
  case Prod(e1, e2) => eval(e1) * eval(e2)
}
→ eval(Numero(1)) + eval(Numero(2))
→ Numero(1) match {
  case Numero(n) => n
  case Suma(e1, e2) => eval(e1) + eval(e2)
  case Prod(e1, e2) => eval(e1) * eval(e2)
} + eval(Numero(2))
→ 1 + eval(Numero(2))
→ 1 + 2
→ 3
```

Listas



- La lista es una **estructura de datos fundamental** para la programación funcional.
- La lista que tiene los elementos x_1, \dots, x_n se escribe en Scala $List(x_1, \dots, x_n)$
- Ejemplos:

```
0  val frutas = List("manzana", "lulo", "guayaba")
1  val numeros = List(1, 2, 3, 4)
2  val IDEI = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
3  val vacia = List()
```

- Dos características importantes de las listas: son **inmutables** y son **recursivas**.

El tipo Lista

- Las listas son **homogéneas**: todos sus elementos deben ser del mismo tipo.
- El tipo de una lista de elementos del tipo T es $\text{scala.List}[T]$ o, en versión corta. $\text{List}[T]$
- Ejemplos:

```
0  scala> val frutas = List("manzana", "lulo", "guayaba")
1    | val numeros = List(1, 2, 3, 4)
2    | val IDEI = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
3    | val vacia = List()
4  val frutas: List[String] = List(manzana, lulo, guayaba)
5  val numeros: List[Int] = List(1, 2, 3, 4)
6  val IDEI: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
7  val vacia: List[Nothing] = List()
```

Los constructores de listas

- Las listas se construyen a partir de dos constructores:
 - La lista vacía *Nil*, y
 - El operador de construcción `::` (se pronuncia *cons*):

X :: xs

construye una lista cuyo primer elemento es *x*, y cuyo resto de elementos es la lista *xs*.

- Ejemplos:

```
0  scala> val frutas = "manzana" :: "lulo" :: "guayaba" :: Nil
1    | val numeros = 1 :: 2 :: 3 :: 4 :: Nil
2    | val IDEI = (1::0::0::Nil) :: (0::1::0::Nil) :: (0::0::1::Nil) :: Nil
3    | val vacia = Nil
4  val frutas: List[String] = List(manzana, lulo , guayaba)
5  val numeros: List[Int] = List(1, 2, 3, 4)
6  val IDEI: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
7  val vacia: collection.immutable.Nil.type = List()
```

Asociatividad a la derecha

- Por convención, los operadores que terminan en ":" son asociativos a derecha, es decir: $A :: B :: C$ se interpreta como $A :: (B :: C)$
Por ejemplo:

```
0 scala> val numeros = 1 :: 2 :: 3 :: 4 :: Nil
1 val numeros: List[Int] = List(1, 2, 3, 4)
```

es lo mismo que escribir:

```
0 scala> val numeros = 1 :: (2 :: (3 :: (4 :: Nil)))
1 val numeros: List[Int] = List(1, 2, 3, 4)
```

- Los operadores que terminan en ":" también son diferentes en cuanto a que ellos son vistos como invocadores de métodos del operando del lado derecho, es decir, el ejemplo de arriba es equivalente a:

```
0 scala> Nil.:::(4).:::(3).:::(2).:::(1)
1 val res5: List[Int] = List(1, 2, 3, 4)
```

Operaciones sobre listas

- Todas las operaciones sobre las listas, se implementan a partir de las constructoras de listas y de las siguientes operaciones selectoras:
 - *head*: devuelve el primer elemento de una lista
 - *tail*: devuelve la lista sin el primer elemento (el *resto* de la lista)
 - *isEmpty*: devuelve *true* si la lista es vacía y *false* si no.
- Todas estas operaciones son métodos de los objetos de tipo Lista:

```
0  scala> frutas.head
1  val res6: String = manzana
2
3  scala> frutas.tail
4  val res7: List[String] = List(lulo, guayaba)
5
6  scala> frutas.tail.head
7  val res8: String = lulo
8
9  scala> frutas.tail.tail.head
10 val res9: String = guayaba
11
12 scala> frutas.tail.tail.tail.head
13 java.util.NoSuchElementException: head of empty list
14
15 scala> frutas.tail.tail.tail.tail
16 java.lang.UnsupportedOperationException: tail of empty list
```

Patrones de listas

- Es posible descomponer las listas con reconocimiento de patrones:
 - *Nil*: el patrón constante *Nil*
 - *p :: ps*: un patrón que hace corresponder la cabeza de la lista con el patrón *p* y la cola de la lista con el patrón *ps*. Este patrón, por ejemplo, no hace correspondencia con una lista vacía.
 - *List(p₁, ..., p_n)*: es una abreviación del patrón
 $p_1 :: p_2 :: \dots :: p_n :: Nil$
- Por ejemplo:
 - *1 :: 2 :: xs*: es un patrón que corresponde con las listas de al menos dos elementos, cuyo primer elemento es el 1 y el segundo es el 2.
 - *x :: Nil*: es un patrón que corresponde con las listas que tienen exactamente un elemento.
 - *List(x)*: idéntico al patrón anterior
 - *List()*: idéntico al patrón *Nil*.
 - *List(2 :: xs)*: es un patrón que corresponde con las listas que tienen exactamente un elemento, y ese elemento es una lista que comienza con 2.

Ordenando listas

- Suponga que se desea ordenar listas de números enteros en orden ascendente:
 - Una forma de ordenar la lista *List(8, 4, 11, 2)* consiste en ordenar primero la cola, es decir la lista *List(4, 11, 2)* y obtener *List(2, 4, 11)*
 - Y a continuación, insertar la cabeza, o sea 8 en la lista *List(2, 4, 11)*, para obtener *List(2, 4, 8, 11)*
- Esta idea describe el *Insertion Sort* u ordenamiento por inserciones.

```
0  def iSort(xs:List[Int]): List[Int] ={
1    def insertar(x:Int , xs:List[Int]): List[Int] = xs match {
2      case List() => List(x)
3      case y :: ys => if (x < y) x :: xs else y :: insertar(x, ys)
4    }
5    xs match {
6      case List() => List()
7      case y :: ys => insertar(y, iSort(ys))
8    }
9  }
10 iSort(List(8, 4, 11, 2))
```

3 7 1 4 2 6
↑
↓
 $i = 0, j=1$

3 7 1 4 2 6
↑
↓
 $i = 1$

1 3 7 4 2 6
↑
↓
 $i = 2$

1 3 4 7 2 6

1 2 3 4 7 6

1 2 3 4 6 7

3 7 1 4 2 6

IS
I

IS([3, 7, 1, 4, 2, 6])

I(3, IS([7, 1, 4, 2, 6]))

I(3, I(7, IS([1, 4, 2, 6])))

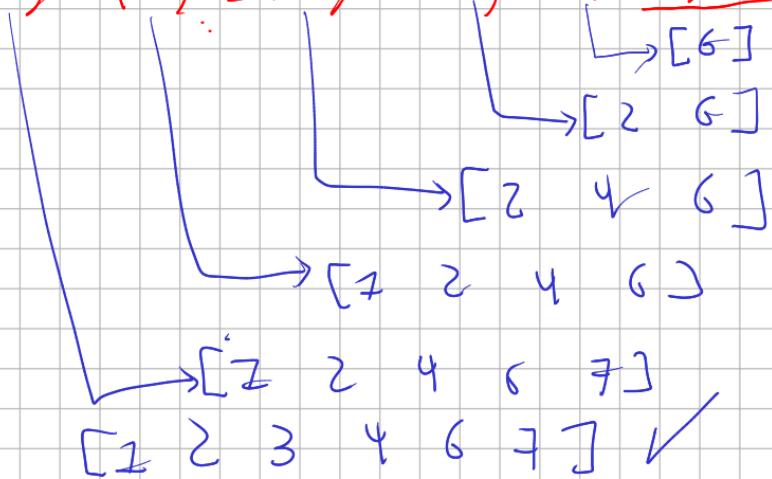
I(3, I(7, I(1, IS([4, 2, 6]))))

I(3, I(7, I(1, I(4, IS([2, 6]))))))

I(3, I(7, I(1, I(4, I(2, IS([6]))))))

I(3, I(7, I(1, I(4, I(2, I(6, IS(NIL)))))))

I(3, I(7, I(1, I(4, I(2, I(6, NIL))))))



Métodos predefinidos (1)

Métodos para acceder a elementos y sublistas:

- `xs.length`: número de elementos de `xs`
- `xs.last`: el último elemento de `xs`; lanza excepción si `xs` es vacía
- `xs.init`: una lista con los mismos elementos de `xs` salvo el último; lanza excepción si `xs` es vacía
- `xs take n`: una lista con los primeros n elementos de `xs` (o `xs` si tiene menos de n elementos)
- `xs drop n`: el resto de la lista `xs` después de quitar los primeros n elementos
- `xs(n)`: el n -ésimo elemento de `xs`; lanza excepción si `xs` no tiene al menos n elementos

Métodos predefinidos (2)

- Creación de nuevas listas:

- $xs \text{ ++ } ys$: la lista que consiste de todos los elementos de xs seguidos de todos los elementos de ys (concatenación)
- $xs.\text{reverse}$: la lista con los mismos elementos de xs pero en orden inverso.
- $xs \text{ updated } (n, x)$: una lista con los mismos elementos de xs salvo el n -ésimo que ahora es x ; lanza excepción si xs no tiene al menos n elementos

- Métodos para buscar elementos:

- $xs \text{ indexOf } x$: devuelve el índice del primer elemento en xs igual a x ; si no aparece x en xs devuelve -1
- $xs \text{ contains } x$: devuelve ($xs \text{ indexOf } x \geq 0$)

Implementando funciones sobre listas: *last* e *init*

- La complejidad de *head* es constante. ¿Qué se puede decir de la complejidad de *last*?
- Miremos una posible implementación de *last* :

```
0 def ultimo[T](xs: List[T]): T = xs match {
1   case List() => throw new Error("ultimo -de- una -lista -vacia")
2   case List(x) => x
3   case _ => ultimo(xs.tail)
4 }
```

Su complejidad es proporcional al tamaño de *xs*

- Cómo implementaría *init* :

```
0 def init[T](xs: List[T]): T = xs match {
1   case List() => throw new Error("lista -inicial -de- una -lista -vacia")
2   case List(x) => ????
3   case _ => ????
4 }
```

Implementando funciones sobre listas: *concat* y *reverse*

- ¿Cómo podríamos implementar la concatenación?

```
0 def concat[T](xs: List[T], ys: List[T]): List[T] = xs match {  
1   case List() => ys  
2   case x::zs => x::concat(zs, ys)  
3 }
```

¿Cuál es su complejidad?

- Cómo implementaría *reverse* :

```
0 def reverse[T](xs: List[T]): List[T] = xs match {  
1   case List() => xs  
2   case List(x) => xs  
3   case x::ys => reverse(ys) ++ List(x)  
4 }
```

¿Cuál es su complejidad? ¿Se podría mejorar?

Implementando funciones sobre listas: *aplanar*

- Las listas no tienen que ser homogéneas. Pueden tener elementos de diferentes tipos mezclados:

```
0 scala> 2::List("a","b")
1 val res0: List[Any] = List(2, a, b)
2
3 scala> List(2, List("lulo", "guayaba"), "nissan")
4 val res1: List[Any] = List(2, List(lulo, guayaba), nissan)
```

- Una función útil cuando se tienen listas de listas, es la que **aplana** la lista, es decir, devuelve la lista con los elementos básicos que no son listas:

```
0 def aplanar(xs: List[Any]): List[Any] = xs match {
1   case List() => xs
2   case y::ys => y match {
3     case List() => aplanar(ys)
4     case z::zs => aplanar(z::zs) ++ aplanar(ys)
5     case _ => y :: aplanar(ys)
6   }
7 }
8 aplanar(List(List(1,1), 2, List(3, List(5,8))))
```

Contenido

1

- La descomposición en el diseño de datos
 - Interpretador de expresiones aritméticas
 - Reconocimiento de patrones
 - Listas y Funciones sobre listas

2

- Pares y tuplas**
 - Ordenamiento por mezclas
 - Pares y tuplas
 - Parametrización de tipos

3

- Funciones de alto orden sobre listas
 - Map
 - Filter
 - Reducción de listas
 - Reverse

Ordenando listas más rápidamente

- Intentaremos definir una función que ordene listas, más eficientemente que el ordenamiento por inserción.
- Un algoritmo conocido para esto se llama el **mergeSort** u ordenamiento por mezclas. La idea es:
 - Si la lista tiene cero o un elementos, ya está ordenada.
 - Sino:
 - Divila la lista original en dos sublistas de tamaño similar (si se puede, igual) cada una conteniendo más o menos la mitad de los mismos elementos de la lista original.
 - Ordene las dos sublistas.
 - Mezcle las dos sublistas ordenadas, en una sola lista ordenada.

Primera implementación de mergeSort

```
0 def msort(xs: List[Int]): List[Int] = {
1   def merge(l1: List[Int], l2: List[Int]): List[Int] = l1 match {
2     case Nil => l2
3     case m::ms => l2 match {
4       case Nil => l1
5       case n::ns => if (m<n) m::merge(ms, l2) else n::merge(l1, ns)
6     }
7   }
8   val n=xs.length/2
9   if (n==0) xs
10  else {
11    val (l1, l2) = xs splitAt n
12    merge(msort(l1), msort(l2))
13  }
14 }
```

La función *splitAt* devuelve dos listas (una hasta antes del elemento *n* y otra del *n*-ésimo elemento en adelante), embebidas en una **pareja** o **tupla** de 2 elementos.

3 7 1 6 2

3 7

1 6 2

1 7

1 6 2

3 7

1 6 2

3 7

6 2

3

6 2

1 2 3

1 2 6

6 7

Pares y tuplas

- Una **pareja** compuesta por x y y se escribe en Scala (x, y) .

```
0 scala> val pareja=("numero", 42)
1   val pareja: (String, Int) = (numero,42)
```

- Las parejas también se pueden usar como patrones:

```
0 scala> val (cadena, valor)=pareja
1   val cadena: String = numero
2   val valor: Int = 42
```

- Funciona de manera análoga con **tuplas** de más de 2 elementos
 - El tipo tupla (T_1, \dots, T_n) es una abreviación del tipo parametrizado

$\text{scala.Tuplen}[T_1, \dots, T_n]$

- Una expresión de tupla (e_1, \dots, e_n) es una abreviación de la aplicación

$\text{scala.Tuplen}(e_1, \dots, e_n)$

- Un patrón de tupla (p_1, \dots, p_n) es una abreviación del patrón

$\text{scala.Tuplen}(p_1, \dots, p_n)$

La clase tupla

- Todas las clases de tuplas se construyen a partir del siguiente patrón de clase:

```
0 case class Tuple2[T1, T2] (_1:T1, _2:T2) {  
1   override def toString = ...  
2 }
```

- Los campos de una tupla se pueden acceder vía `_1`, `_2`, ...
En lugar del reconocimiento de patrones

```
0 scala> val (cadena, valor)=pareja  
1 val cadena: String = numero  
2 val valor: Int = 42
```

se pudo haber escrito:

```
0 scala> val cadena=pareja._1  
1 val cadena: String = numero  
2 scala> val valor=pareja._2  
3 val valor: Int = 42
```

Pero se prefiere escribir con el reconocimiento de patrones

Reescribiendo *merge*

- Tal como escribimos *merge* se usa reconocimiento de patrones anidado. Esto no refleja la simetría del *merge*.
- Reescriba *merge* usando reconocimiento de patrones sobre parejas:

```
0 def merge(l1: List[Int], l2: List[Int]): List[Int] = (l1, l2) match {  
1   ???  
2 }
```

Haciendo un ordenamiento más general

- ¿Cómo parametrizar *msort* de manera que ordene listas de cualquier tipo?
- Nótese que no es suficiente parametrizar *msort*:

```
0 def msort[T](xs: List[T]): List[T] = ...
```

porque *msort* usa *merge* y esta usa $<$ no es un comparador del tipo T .

- **Idea:** parametrizar con la función de comparación del tipo T
- Diseñamos *msort* **polimórfica** pasando el operador de comparación:

```
0 def msort[T](xs: List[T])(mq:(T,T)⇒Boolean): List[T] = {
1   def merge(l1: List[T], l2: List[T]): List[T] = (l1, l2) match {
2     case (Nil, _) => l2
3     case (_, Nil) => l1
4     case (m :: ms, n :: ns) =>
5       if (mq(m,n)) m :: merge(ms, l2) else n :: merge(l1, ns)
6   }
7   val n=xs.length/2
8   if (n==0) xs
9   else {
10    val (l1,l2) = xs splitAt n
11    merge(msort(l1)(mq), msort(l2)(mq))
12  }
13 }
14 msort(List(1,3,2, 4, 7, 5))((x:Int, y:Int)⇒ (x<y))
15 msort(List("a","c","f", "e", "d", "b"))((x:String, y:String)⇒ (x<y))
```

Contenido

1

- La descomposición en el diseño de datos
- Interpretador de expresiones aritméticas
 - Reconocimiento de patrones
 - Listas y Funciones sobre listas

2

- Pares y tuplas
- Ordenamiento por mezclas
 - Pares y tuplas
 - Parametrización de tipos

3

- Funciones de alto orden sobre listas
- Map
 - Filter
 - Reducción de listas
 - Reverse

Patrones recurrentes en computaciones sobre listas

- A partir de los ejemplos anteriores, podemos ver que las funciones sobre listas tienen con frecuencia estructuras similares.
- Se pueden identificar patrones recurrentes:
 - Transformar cada elemento de la lista de cierta manera
 - Recuperar los elementos de la lista que satisfagan algún criterio
 - Combinar los elementos de la lista usando un operador
- Los lenguajes funcionales permiten a los programadores escribir funciones genéricas que implementan este tipo de patrones por medio de **funciones de alto orden**

Aplicando una función a los elementos de una lista: *map*

- Una operación común consiste en transformar cada elemento de una lista y devolver la lista de los resultados de esas transformaciones.
- Por ejemplo, multiplicar todos los elementos de una lista por un mismo factor:

```
0 def escalarLista(xs: List[Double], factor: Double): List[Double] = xs match {
1   case Nil => Nil
2   case y :: ys => y * factor :: escalarLista(ys, factor)
3 }
```

- Este esquema se generaliza por medio del método *map* de las listas:

```
0 abstract class List[T] { ... }
1   def map[U](f: T => U): List[U] = this match{
2     case Nil => this
3     case x :: xs => f(x) :: xs.map(f)
4   }
5 ...
6 }
```

- Usando *map*, se puede escribir *escalarLista* más sencillo:

```
0 def escalarLista(xs: List[Double], factor: Double): List[Double] = xs map (_ => _ * factor)
```

Ejercicio

Escriba una función *elevarCuadrado* que tome una lista de números y devuelva la lista con los cuadrados de esos números.

- Haga una primera versión directa sin usar *map*:

```
0 def elevarCuadrado(xs: List[Double]): List[Double] = xs match {  
1   case Nil => ???  
2   case y :: ys => ???}
```

- Haga una segunda versión usando *map*:

```
0 def elevarCuadrado2(xs: List[Double]): List[Double] = xs map (???)
```

Filtrando los elementos de una lista: *filter*

- Otra operación común consiste en seleccionar cada elemento de una lista que cumpla una condición.
- Por ejemplo, seleccionar los números positivos de una lista:

```
0 def positivosLista(xs: List[Double]): List[Double] = xs match {
1   case Nil => Nil
2   case y::ys => if (y>0) y:: positivosLista(ys) else positivosLista(ys)
3 }
```

- Este esquema se generaliza por medio del método *filter* de las listas:

```
0 abstract class List[T] { ...
1   def filter(p: T=>Boolean): List[T] = this match{
2     case Nil => this
3     case x::xs => if (p(x)) x::xs.filter(p) else xs.filter(p)
4   }
5   ...
6 }
```

- Usando *filter*, se puede escribir *positivosLista* más sencillo:

```
0 def positivosLista(xs: List[Double]): List[Double] = xs filter (x=>(x>0))
```

Variaciones de *filter*

Además de *filter* hay otros métodos que extraen sublistas basados en predicados:

- $xs \text{ filterNot } p$: igual que $xs \text{ filter}(x \Rightarrow !p(x))$
- $xs \text{ partition } p$: igual que $(xs \text{ filter}(p), xs \text{ filterNot}(p))$ pero calculado en una sola pasada
- $xs \text{ takeWhile } p$: devuelve el prefijo más largo de xs con elementos que satisfagan p .
- $xs \text{ dropWhile } p$: devuelve el resto de la lista xs después de eliminar el prefijo más largo de xs con elementos que satisfagan p .
- $xs \text{ span } p$: igual que $(xs \text{ takeWhile}(p), xs \text{ dropWhile}(p))$ pero calculado en una sola pasada

Ejercicios

- Escriba una función *empaquetar* que dada una lista de elementos, empaquete los elementos duplicados consecutivos en una sublista, y devuelva la lista de sublistas respectiva.

empaquetar(List("a", "a", "a", "b", "c", "c", "a"))

deve devolver

List(List("a", "a", "a"), List("b"), List("c", "c"), List("a"))

- Usando *empaquetar* escriba la función *codificar* que dada una lista de elementos, devuelve una lista de parejas (*elemento numero*) indicando el *numero* de veces consecutivas que se repite *elemento* en la lista.

codificar(List("a", "a", "a", "b", "c", "c", "a"))

deve devolver

List(("a", 3), ("b", 1), ("c", 2), ("a", 1))

Reducción de listas

- Otra operación útil sobre las listas consiste en **combinar** los elementos de una lista usando un **operador binario** dado.

$$\text{suma}(\text{List}(x_1, \dots, x_n)) = 0 + x_1 + \dots + x_n$$

$$\text{prod}(\text{List}(x_1, \dots, x_n)) = 1 * x_1 * \dots * x_n$$

- Podemos implementarlos con los esquemas recursivos usuales:

```
0 def suma(xs: List[Int]): Int = xs match {
1   case Nil => 0
2   case y :: ys => y + suma(ys)
3 }
4 def prod(xs: List[Int]): Int = xs match {
5   case Nil => 1
6   case y :: ys => y * prod(ys)
7 }
```

Reducción a la izquierda

- Esos esquemas se pueden abstraer usando el método genérico *reduceLeft*, el cual aplica un operador binario a los diferentes elementos de una lista, asociando a izquierda:

$List(x_1, \dots, x_n) \text{ reduceLeft } op = (\dots ((x_1 \underset{\text{red}}{\text{op}} x_2) \underset{\text{red}}{\text{op}} x_3) \underset{\text{red}}{\text{op}} \dots) \underset{\text{red}}{\text{op}} x_n$

- Usando *reduceLeft* podemos implementar *suma* y *prod* de la siguiente manera:

```
0 def suma(xs: List[Int]): Int = (0::xs).reduceLeft ((x,y) => x+y)
1 def prod(xs: List[Int]): Int = (1::xs).reduceLeft ((x,y) => x*y)
```

o de manera más corta:

```
0 def suma(xs: List[Int]): Int = (0::xs).reduceLeft (- + -)
1 def prod(xs: List[Int]): Int = (1::xs).reduceLeft (- * -)
```

foldLeft

- Hay una función más general denominada *foldLeft*, la cual aplica un operador binario a los diferentes elementos de una lista, asociando a izquierda, pero si la lista es vacía devuelve un **acumulador**. *reduceLeft* no está definida sobre listas vacías.

$$(List(x_1, \dots, x_n) \text{ foldLeft } z)(op) = (\dots ((z \text{ op } x_1) \text{ op } x_2) \text{ op } \dots) \text{ op } x_n$$

$$(List() \text{ foldLeft } z)(op) = z$$

- Usando *foldLeft* podemos implementar *suma* y *prod* de la siguiente manera:

```
0 def suma(xs: List[Int]): Int = (xs foldLeft 0) ((x,y) => x+y)
1 def prod(xs: List[Int]): Int = (xs foldLeft 1) ((x,y) => x*y)
```

o de manera más corta:

```
0 def suma(xs: List[Int]): Int = (xs foldLeft 0) (- + -)
1 def prod(xs: List[Int]): Int = (xs foldLeft 1) (- * -)
```

Implementación de ReduceLeft y FoldLeft

foldLeft y *reduceLeft* están implementadas en la clase *Lista*:

```
0 abstract class List[T] { ...  
1     def reduceLeft(op: (T, T)⇒ T): T = this match {  
2         case Nil => throw new Error("Nil.reduceLeft")  
3         case x::xs => (xs foldLeft x) (op)  
4     }  
5     def foldLeft[U](z:U) (op: (U, T)⇒ U): U = this match {  
6         case Nil => z  
7         case x::xs => (xs foldLeft op(z,x)) (op)  
8     }  
9     ...  
10 }
```

ReduceRight y FoldRight

De manera análoga, se pueden definir las funciones que asocian hacia la derecha:

- *reduceRight* y *foldRight* aplican el operador binario asociando a la derecha:

$$\text{List}(x_1, \dots, x_n) \text{ reduceRight } op = x_1 \text{ op } (\dots (x_{n-2} \text{ op } (x_{n-1} \text{ op } x_n)) \dots)$$

$$(\text{List}(x_1, \dots, x_n) \text{ foldRight } z)(op) = x_1 \text{ op } (\dots (x_{n-1} \text{ op } (x_n \text{ op } z)) \dots)$$

$$(\text{List}()) \text{ foldRight } z)(op) = z$$

- La implementación en la clase *List* sería:

```
0 abstract class List[T] { ...  
1     def reduceRight(op: (T, T)⇒ T): T = this match {  
2         case Nil => throw new Error("Nil.reduceRight")  
3         case x::Nil => x  
4         case x::xs => op(x, xs.reduceRight(op))  
5     }  
6     def foldRight[U](z:U) (op: (T, U)⇒ U): U = this match {  
7         case Nil => z  
8         case x::xs => op(x, (xs foldRight z)(op))  
9     }  
10    ...  
11 }
```

¿Cuál es la diferencia entre FoldLeft y FoldRight?

- Si los operadores binarios utilizados son conmutativos y asociativos, *foldLeft* y *foldRight* tendrán el mismo resultado. Pero existe una diferencia en eficiencia. **¿Cuál es más eficiente?**
- En caso contrario, sólo uno de los dos funciona correctamente. Por ejemplo, considere la siguiente versión de concatenación de listas:

```
0 def concat(xs: List[T], ys: List[T]): List[T] =  
1   (xs foldRight ys) (_ :: _)
```

¿Es posible reemplazar *foldRight* por *foldLeft*? ¿Por qué?

Invirtiendo listas eficientemente

- Se quiere desarrollar una función *reverse* que invierta una lista en orden lineal.
- Rellene el siguiente programa que usa *foldLeft*:

```
0  def reverse[T](xs: List[T]): List[T] =  
1    (xs foldLeft z) (op)
```

¿Cuál debe ser el valor de *z*? ¿Cuál el valor de *op*?

- Infiera *z* del hecho que $\text{reverse}(\text{Nil}) == \text{Nil}$
- Infiera *op* del hecho que $\text{reverse}(\text{List}(x)) == \text{List}(x)$