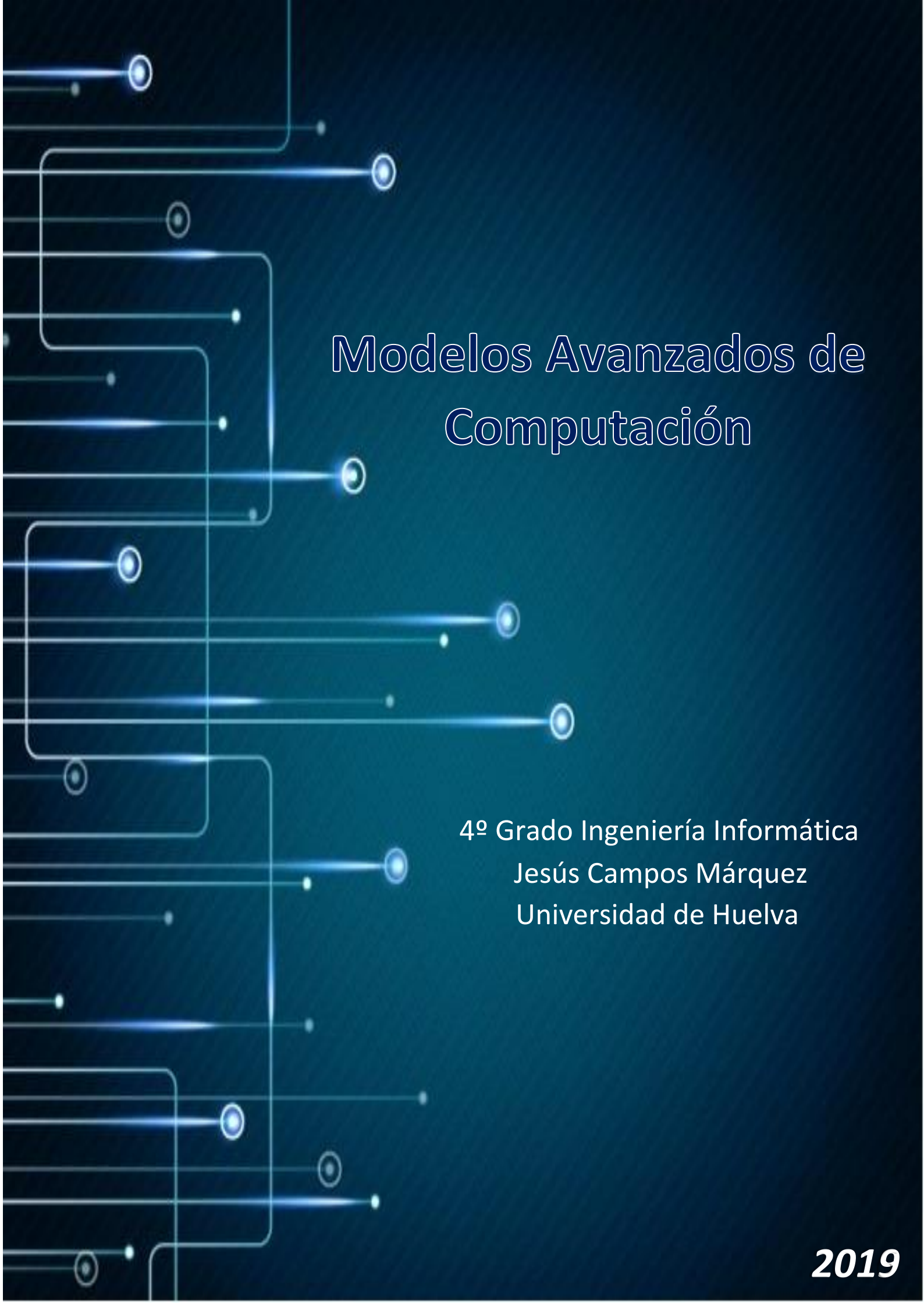


Modelos Avanzados de Computación

4º Grado Ingeniería Informática
Jesús Campos Márquez
Universidad de Huelva

2019

Índice

Definición de funciones

1. *Definición de funciones en Haskell*

Se crea un .hs en el que se pone el nombre de la función a llamar y a que es igual. Devuelve un tipo definido por Haskell si no se especifica lo contrario

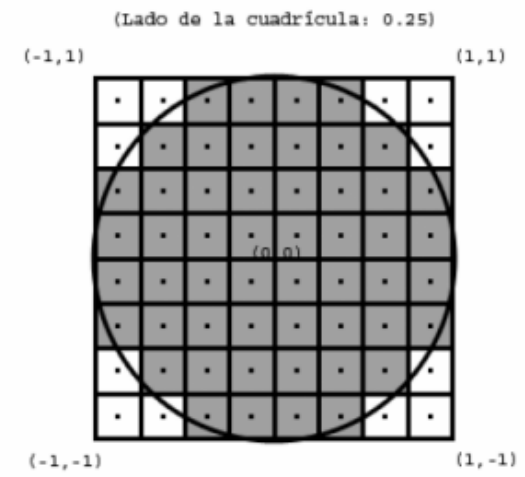
2. *Definición de funciones en SCALA*

```
def nombreFuncion (nomParametro: tipo): Tipo que devuelve ={

        CODIGO
}
```

3. Ejercicio de ejemplo:

2. (Febrero 2006) Implementar una función que aproxime el valor de *pi* . Para obtener el valor aproximado de *pi* utilizaremos un cuadrado de lado 2. En el centro del cuadrado fijaremos en centro de referencia (0,0). Haremos una rejilla dentro del cuadrado de lado *t*. Por último, contaremos cuantos centros de los cuadrados de la rejilla están dentro del círculo. Esto nos dará un valor aproximado de *pi*. Cuanto menor sea *t*, más precisión tendrá la aproximación.



Área del círculo:

$A = \pi * r^2$

Distancia entre los puntos (x1,y1) y (x2,y2):

$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

```
/* EN HASKELL
  --t es un lado del cuadradito
  --[(x,y) | x<-[-1+t/2,-1+t/2+t..1-t/2], y<-[-1+t/2,-1+t/2+t..1-t/2]]

  --la funcion seria asi
 aprox_pi t = t^2*fromIntegral (length [(x,y) | x<-[-1+t/2,-1+t/2+t .. 1-t/2], y<-[-1+t/2,-1+t/2+t .. 1-t/2], distancia x y <=1])
 -- la lamada seria aprox_pi 0.25 dando todas las coordenadas de los puntos d ela imagen.
 -- tenemos que quedarnos con aquellos cuya distancia desde el centro al punto este comprendida en un radio
   where distancia x y = sqrt (x^2 + y^2)
*/
```


Nombre de la función Parámetro que se le pasa Operaciones en la función

```
def aprox_pi(t: Double): Double ={
  //x<-[-1+t/2,-1+t/2+t .. 1-t/2], y<-[-1+t/2,-1+t/2+t .. 1-t/2], distancia x y <=1])
  var x=((BigDecimal((-1+t/2+t)-t) to BigDecimal(1-t/2)) by BigDecimal((-1+t/2+t)-(-1+t/2))).toList
  println(x)
  var y=((BigDecimal((-1+t/2+t)-t) to BigDecimal(1-t/2)) by BigDecimal((-1+t/2+t)-(-1+t/2))).toList
  println(y)
  var listafinal=for{i<-x;j<-y
    if(distancia(i.toDouble,j.toDouble)<=1)
  } yield(i, j)
  println(listafinal)
  var res=(t*t)*listafinal.length
  return res

}

def distancia(x: Double, y: Double): Double ={

  return Math.sqrt((x*x)+(y*y))
}

def main(args: Array[String]): Unit = {
  println("Resultado: "+aprox_pi(0.25))
}
```

Prioridad de operadores

1. En Haskell

Todas las funciones tienen precedencia máxima, denotada por el número 9 para su precedencia.
Las precedencias de los principales operadores son las siguientes:

9: .
8: **
7: *, /, `div`, `mod`
6: +, -
5: ++,:
4: ==, /=, <, <=, >, >=
3: &&
2: ||
1: >>, >>=

Nótese que `div` y `mod` tienen precedencia 7, mientras que div y mod tienen precedencia 9 (como todas las funciones).

En una expresión, a igualdad de precedencia entre dos operadores distintos, se realiza primero el que se encuentre más a la izquierda. Por ejemplo, en 7-8+2 se computa 1, pues se hace antes la resta.
Si el mismo operador se utiliza varias veces seguidas en una misma expresión, asociará a izquierdas o a derechas dependiendo de cómo se haya definido.

2. En SCALA

Tipo de operador	operadores	dirección de encuadernación
evaluación de la expresión	() []. Expr ++ expr--	De izquierda a derecha
operador unitario	* Y + - ~ ++ Expr --expr * /% + - >> << <> = <=> ==! =	De derecha a izquierda
Los operadores bit a bit	y ^ | && ||	De izquierda a derecha
operador ternario	?:	De derecha a izquierda
Operadores de asignación	= + = - = * = / =% = >> << = = & = ^ = | =	De derecha a izquierda
coma	,	De izquierda a derecha

Evaluación perezosa

1. **¿Qué es la evaluación perezosa?**

   Es cuando pasamos por parámetro una parámetro mediante una operación. Por ejemplo: supongamos que tengo la función multiplicación mult(a, b) = a * a

   Si llamara a esta función con los parámetros 8 y 5+2 de forma tradicional sucedería lo siguiente:

   mult(8, 5+2)

   mult(8,7)

   8*8

   64

   Pero si ejecutamos el código de forma perezosa:

   mult(8, 5+2)

   8 * 8

   64

   En este caso la ejecución perezosa fue más eficiente que la forma tradicional.

   En Haskell sería algo como:

   - Definicion dela función

     ```
     mult' :: Int -> Int -> Int
     mult' x = \y -> x*y
     ```

   - Desglosado sería:

     ```
     mult' (1+2) (2+3)
     = mult' 3 (2+3)      [por def. de +]
     = (\y -> 3*y) (2+3)  [por def. de mult']
     = (\y -> 3*y) 5      [por def. de +]
     = 3*5                [por def. de +]
     = 15                 [por def. de *]
     ```

Funciones de orden superior y Currificación

1. **¿Qué es una función de orden superior?, ¿Y la currificación?**
   Las funciones de Haskell pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama función de orden superior.
   Por otro lado la como Haskell solo puede recibir un parámetro, se utiliza la llamada currificación que hace como dos llamas a la misma función, esto se puede ver en el ejemplo siguiente de forma más clara.

2. **En Haskell:**
   Al aplicar max 4 5 primero se crea una función que toma un solo parámetro y devuelve 4 o el parámetro, dependiendo de cual sea mayor. Luego, 5 es aplicado a esa función y esta produce el resultado deseado. Las siguientes dos llamadas son equivalentes:
   - **Max 4 5**
   - **(Max 4) 5**

3. **En SCALA:**
   Estas funciones son las que *toman otras funciones como parámetros*, o las cuales *el resultado es una función*. Por ejemplo, una función apply la cual toma otra función f y un valor v como parámetros y aplica la función f a v:
   - **def apply(f: Int => String, v: Int) = f(v)**

Composición de funciones

1. **¿Qué es una composición de funciones?**
   En matemáticas la composición de funciones está definida como $(f \circ g)x = f(g(x))$ que significa que al componer dos funciones se crea una nueva que, cuando se llama con un parámetro, digamos x, es equivalente a llamar a g con x y luego llamar a f con el resultado anterior.

2. **En Haskell**
   En Haskell la composición de funciones es prácticamente lo mismo. Realizamos la composición de funciones con la función. Por ejemplo, digamos que tenemos una lista de números y queremos convertirlos todos en negativos. Una forma de hacerlo sería obteniendo primero el número absoluto y luego negándolo, algo así:

   map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
   [-5,-3,-6,-7,-3,-2,-19,-24]

3. **En SCALA**
   Scala tiene 2 operadores para componer funciones: ***compose*** y ***andThen.*** La diferencia es que mientras f compose g es f(g(x)), f andThen g es g(f(x)). Por tanto, si queremos definir una función que primero convierta la cadena recibida en entero y después le sume uno a ese entero, podemos hacerlo de dos formas:
   val composed1 = addOne _ compose toInt
   val composed2 = toInt _ andThen addOne

Listas

1. **En Haskell**
   En haskell utilizamos las llamadas listas intensionales. Estas son muy similares a los conjuntos definidos de forma intensiva. En el caso de que queramos el doble d ellos 10 primeros numeros, la lista intensional que deberíamos usar sería [x*2 | x <- [1..10]]. x es extraído de [1..10] y para cada elemento de [1..10] (que hemos ligado a x) calculamos su doble. Su resultado es:

   [x*2 | x <- [1..10]]
   [2,4,6,8,10,12,14,16,18,20]

2. **En SCALA**
   Para hacer una listas intensional en SCALA se utiliza el yield, que crea listas por comprensión. . Las comprensiones tienen la forma `for (enumeradores) yield e`, donde `enumeradores` se refiere a una lista de enumeradores separados por el símbolo punto y coma (;). Un *enumerador* puede ser tanto un generador el cual introduce nuevas variables, o un filtro. La comprensión evalúa el cuerpo `e` por cada paso (o ciclo) generado por los enumeradores y retorna una secuencia de estos valores. El resultado del ejemplo anterior en SCALA es:

   ```
   var numeros=(1 to 10).toList

    def num1(): Unit ={
      var listafinal= for{i<-numeros
                  if(i<=5)} yield (i*2)
      println(listafinal)
   }
   ```

Tuplas

1. **¿Qué es una tupla?**
   De alguna forma, las tuplas son parecidas a las listas. Ambas son una forma de almacenar varios valores en un solo valor. Sin embargo, hay unas cuantas diferencias fundamentales. Una lista de números es una lista de números. Ese es su tipo y no importa si tiene un sólo elemento o una cantidad infinita de ellos. Las tuplas sin embargo, son utilizadas cuando sabes exactamente cuantos valores tienen que ser combinados y su tipo depende de cuantos componentes tengan y del tipo de estos componentes. Las tuplas se denotan con paréntesis y sus valores se separan con comas.

2. **En Haskell**

   [((x*3),x<=3) | x <-[1..10],x<7]
   El resultado sería: [(3,True),(6,True),(9,True),(12,False),(15,False),(18,False)]

3. **En SCALA**

```
def num5(): Unit ={
   var listafinal=for{i<-numeros
     if(i<7)} yield (i*3,i<=3)
   println(listafinal)
 }
```

Recursividad

1. **¿Qué es la recursividad o recursión?**
   Si aún no sabes que es la recursión, lee esta frase: La recursión es en realidad una forma de definir funciones en la que dicha función es utiliza en la propia definición de la función. Las definiciones matemáticas normalmente están definidas de forma recursiva. Por ejemplo, la serie de Fibonacci se define recursivamente. Primero, definimos los dos primeros números de Fibonacci de forma no recursiva. Decimos que F(0) = 0 y F(1) = 1, que significa que el 1º y el 2º número de Fibonacci es 0 y 1, respectivamente. Luego, para cualquier otro índice, el número de Fibonacci es la suma de los dos números de Fibonacci anteriores. Así que F(n) = F(n-1) + F(n-2). De esta forma, F(3) = F(2) + F(1) que es F(3) = (F(1) + F(0)) + F(1).

2. **En Haskell**

```
factorial 0 = 1
factorial m(n+1)=m*factorial n
```

3. **En SCALA**

```
def factorial(x: BigInt): BigInt =
    if (x == 0) 1 else x * factorial(x - 1)
```

Más sobre SCALA

1. ¿Por qué utilizar SCALA?
   Escalabilidad, Funcionalidad, Orientación a objetos, Tipado estático, Extensible, Productivo, Interoperabilidad con java, Open source.
2. IDE's que trabajan con SCALA
   IntellJ, ENSIDE, NetBeans, Scala IDE for Eclipse.
3. Empresas que utilizan SCALA
   Everis, Entelgy, Keapps, Ibertech, etc.

Bibliografía

https://docs.scala-lang.org/tutorials/FAQ/yield.html
https://www.tutorialspoint.com/scala/index.htm
https://docs.scala-lang.org/es/tutorials/scala-for-java-programmers.html
http://aprendehaskell.es/content/Empezando.html
http://antares.sip.ucm.es/~fernando/pf/temas/precedencia.html
http://www.w3big.com/es/scala/scala-operators.html
https://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-10.html
http://www.w3big.com/es/scala/higher-order-functions.html
http://aprendehaskell.es/content/OrdenSuperior.html#funciones-currificadas
https://docs.scala-lang.org/es/tutorials/tour/sequence-comprehensions.html
http://manuel.midoriparadise.com/2014/11/composicion-de-funciones-en-scala/