

PROGRAMACIÓN LÓGICA Y FUNCIONAL

SESIÓN 07

CICLO: AGOSTO 2021



Universidad
Tecnológica
del Perú

CONTENIDO

RECURSIVIDAD CON LISTAS



Pautas de trabajo

- Los días que tengamos clases debemos conectarnos a través de Zoom.
- La participación de los estudiantes se dará través del **chat de Zoom.**
- En Canvas encontrarán la clase de hoy, el ppt de la sesión 7, Laboratorio 7

RECORDANDO LA SESIÓN ANTERIOR

¿ LISTAS , APLICACIONES DE LISTAS?

Levantemos la mano para participar



Logro del Aprendizaje

Al finalizar la presente sesión el estudiante utiliza el concepto de listas y recursividad para generar programas



Utilidad

¿Porqué será importante realizar algoritmos de recursión utilizando listas?



RECUSIÓN

La recursión es un concepto muy utilizado en programación. Se basa en expresar el resultado de un problema como operaciones aplicadas sobre una instancia reducida del mismo problema, hasta que se llega a un caso donde el problema queda bien definido. Este concepto es el mismo que se utiliza al demostrar inductivamente un problema.



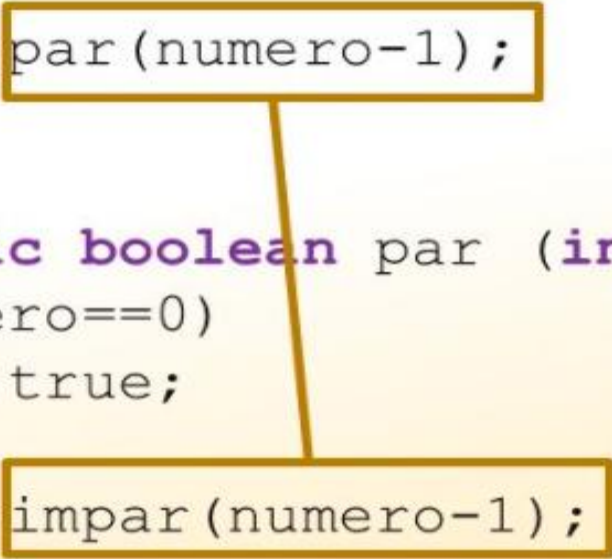
*“Para entender la
recursividad,
primero hay que
entender lo qué
es la **recursividad**”*



Recursividad indirecta o mutua

```
public static boolean impar (int numero) {  
    if (numero==0)  
        return false;  
    else  
        return par(numero-1);  
}
```

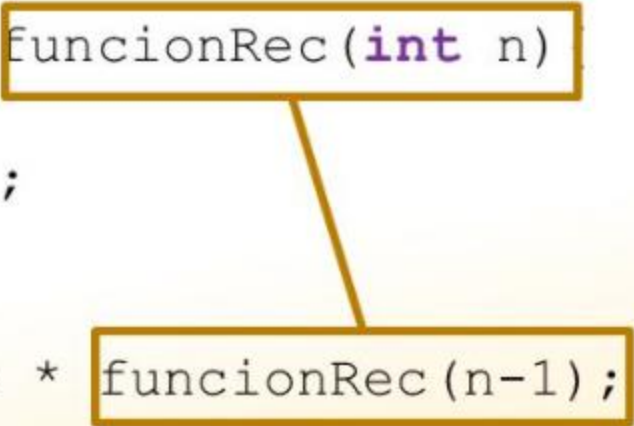
```
public static boolean par (int numero) {  
    if (numero==0)  
        return true;  
    else  
        return impar(numero-1);  
}
```



Se llaman entre ellas

Recursividad directa

```
public static int funcionRec(int n)
    if (n == 0) {
        return 1;
    }
    else {
        return n * funcionRec(n-1);
    }
}
```



Se llama a sí misma directamente

Para que una función pueda ser
recursiva tienen que cumplirse
ciertos requisitos:

Que exista
un criterio de finalización o “caso base”



Lo definimos matemáticamente

Solución iterativa:

$$x! = 1 \cdot 2 \cdot \dots \cdot (x-1) \cdot x$$

Solución recursiva:

$$\text{Si } x = 0 \rightarrow x! = 1$$

$$\text{Si } x > 0 \rightarrow x! = x \cdot (x-1)!$$

Solución **iterativa** en Java

```
public static int factorial(int n){  
    int fact = 1;  
    for (int i = 1 ; i <= n ; i++){  
        fact *= i;  
    }  
    return fact;  
}
```

RECUSIÓN

A modo de ejemplo, veremos la función parcial recursiva que calcula el factorial de un natural.

$F: \mathbb{N} \rightarrow \mathbb{N}$

$$F(x) = \begin{cases} 1 & \text{si } x = 0 \\ x * F(x-1) & \text{si } x > 0 \end{cases}$$

Se puede observar que para computar el resultado de $F(x)$ (en caso que sea mayor que 0), es necesario calcular el resultado de $F(x-1)$ y así sucesivamente hasta llegar al caso base ($x = 0$). Es necesario que estos casos estén definidos para asegurar que la recursión termine.

No siempre el caso base de una función recursiva sucede cuando $x=0$, ni siempre se reduce el x en 1 en el paso recursivo. Lo que se debe garantizar es que para todo x perteneciente al dominio de la función, tras aplicar una cantidad finita de pasos recursivos, se alcanza al caso base.

Otro ejemplo que utilizaremos para ilustrar la recursión, es con el cálculo recursivo de la serie de Fibonacci.

$F: \mathbb{N} \rightarrow \mathbb{N}$

$$F(x) = \begin{cases} 1 & \text{si } x < 2 \\ F(x-1) + F(x-2) & \text{si } x \geq 2 \end{cases}$$

Como podemos ver, en este ejemplo, el caso base puede ser necesario definirlo para más de un natural.

RECUSIÓN

Funciones Recursivas en Haskell

Aplicaremos este concepto en el lenguaje funcional Haskell, el cual ya fue presentado en la clase anterior. La función Factorial se declararía y definiría en sintaxis de Gofer, de la siguiente manera:

```
factorial :: Int -> Int
factorial 0 = 1                (1)
factorial x = x * (factorial (x-1)) (2)
```

A modo de ejemplo, haremos el seguimiento del cálculo de ésta función aplicada al natural 3.

Invocación: factorial 3

=> 3 * (factorial (3-1))	reducción por definición 2
=> 3 * (factorial 2)	reducción por función (-)
=> 3 * (2 * (factorial (2-1)))	reducción por definición 2
=> 3 * (2 * (factorial 1))	reducción por función (-)
=> 3 * (2 * (1 * factorial (1-1)))	reducción por definición 2
=> 3 * (2 * (1 * factorial 0))	reducción por función (-)
=> 3 * (2 * (1 * 1))	reducción por definición 1
=> 3 * (2 * 1)	reducción por función (*)
=> 3 * 2	reducción por función (*)
=> 6	reducción por función (*)

RECUSIÓN

Pattern Matching (coincidencia de patrón): identificar una sección del resultado parcial que se esta evaluando que satisface con alguna de las definiciones ya establecidas. Por ejemplo cuando se evalúa

`3 * (factorial 2)`

el interprete hace Pattern Matching de factorial 2 con la definición factorial x ya que en la declaración de esta función se especificó que el parámetro de la función factorial es de tipo Int.

Reducción: Una vez que se hace el Pattern Matching, se reemplaza por lo que establece definición, instanciando las variables de la definición por los valores con los que fue invocada. En nuestro ejemplo:

`3 * (factorial 2) => (reducción por factorial x con x=2) =>`
`=> 3 * (x * (factorial (x-1))) | x=2 => instancia x en 2 => 3 * (2 * (factorial (2-1)))`

Definiremos ahora la función fibonacci en Haskell.

```
fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci x = (fibonacci (x-1)) + (fibonacci (x-2))
```

RECUSIÓN

Se dice que dos números naturales a y b son "amigos" si la suma de los divisores positivos de a menores que a es igual a la suma de los divisores positivos de b menores que b . Definamos una función en gofer que determine si dos números son amigos.

```
amigos :: Int -> Int -> Bool
amigos a b = (sdp a) == (sdp b)

-- Esta función devuelve la suma de divisores positivos

sdp :: Int -> Int
sdp x = sdpAux 1 x

sdpAux :: Int -> Int -> Int
sdpAux n y | n==y = 0
            | n `divide` y = n + (sdpAux (n+1) y)
            | otherwise = sdpAux (n+1) y

divide :: Int -> Int -> Bool
divide a b = (b `mod` a) == 0
```

EJERCICIOS

```
{-  
Ejercicio 7.1  
Incrementa todos los elementos de una lista de enteros.  
-}
```

```
incLista :: [Integer] -> [Integer]  
incLista [] = []  
incLista (x:xs) = (x + 1) : incLista xs
```

```
{-  
Ejercicio 7.2  
Función equivalente a 'zip', empareja dos listas.  
-}
```

```
emparejaListas :: [a] -> [a] -> [(a,a)]  
emparejaListas [] _ = []  
emparejaListas _ [] = []  
emparejaListas (x:xs) (y:ys) = (x,y) : (emparejaListas xs ys)
```

EJERCICIOS

```
{-  
Ejercicio 7.3  
Resto de forma recursiva utilizando sustracciones.  
-}
```

```
resto :: Integer -> Integer -> Integer  
resto x y  
  | x < y = x  
  | otherwise = resto (x - y) y
```

```
{-  
Ejercicio 7.4  
Cociente de forma recursiva utilizando sumas y restas.  
-}
```

```
cociente :: Integer -> Integer -> Integer  
cociente x y  
  | x < y = 0  
  | otherwise = 1 + cociente (x - y) y
```



EJERCICIOS

```
{-  
Ejercicio 7.5  
Función recursiva que devuelve el sumatorio desde un valor entero hasta otro.  
-}
```

```
sumatorio :: Integer -> Integer -> Integer  
sumatorio a b  
  | a < b = a + sumatorio (a + 1) b  
  | a == b = b  
  | otherwise = error "el primer argumento es mayor que el segundo"
```

```
{-  
Ejercicio 7.6  
Función recursiva que calcula el producto de los números que hay entre el  
primer y segundo argumento, ambos incluidos.  
-}
```

```
productoIntervalo :: Integer -> Integer -> Integer  
productoIntervalo a b  
  | a < b = a * productoIntervalo (a + 1) b  
  | a == b = b  
  | otherwise = error "el primer argumento es mayor que el segundo"
```

EJERCICIOS

```
{-
Ejercicio 7.7
Función que calculan números combinatorios.
-}

-- ( m n ) = m! / (m - n)! * n!
combinatorios :: Integer -> Integer -> Integer
combinatorios _ 0 = 1
combinatorios m n
    | m == n = 1
    | otherwise = div (factorial m) ((factorial (m - n)) * (factorial n))

{-
Ejercicio 7.8
Función que determina si un número positivo de cuatro cifras es capicúa
-}

-- Sean las cifras abcd, compara (a == d) && (b == c)

esCapicua :: Integer -> Bool
esCapicua x
    | x > 9999 = error "argumento mayor que 9999"
    | x > 999 = ((div x 1000) == (mod x 10)) &&
        ((mod (div x 100) 10) == (div (mod x 100) 10))
    | otherwise = error "argumento menor que 999"
```

EJERCICIOS

```
{-
```

```
Ejercicio 7.9
```

```
Función que calcula la suma de las cifras de un número natural.
```

```
-}
```

```
sumaCifras :: Integer -> Integer
```

```
sumaCifras x
```

```
  | x < 10 = x
```

```
  | otherwise = (mod x 10) + sumaCifras (div x 10)
```

```
{-
```

```
Ejercicio 7.10
```

```
Funciones que transforman una lista de dígitos en su correspondiente valor entero y viceversa.
```

```
-}
```

```
aEntero :: [Integer] -> Integer
```

```
aEntero ls = case ls of
```

```
  [] -> 0
```

```
  x:xs -> x*10^(length xs) + aEntero xs
```

```
aLista :: Integer -> [Integer]
```

```
aLista x
```

```
  | x < 10 = [x]
```

```
  | otherwise = aLista (div x 10) ++ [(mod x 10)]
```

Actividad grupal



LABORATORIO 07



Qué hemos aprendido el día de hoy?



Utiliza el chat para participar





**Universidad
Tecnológica
del Perú**