

Concepto de Recursión

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.

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.

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 (*)
```

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

```
3 * (factorial 2)
```

el intérprete 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))
```

Problema

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
```