Recursión

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Recursión

- Definir algo en términos de sí mismo.

  En la definición aparece el elemento que se está definiendo
- Existen uno o varios casos base
- La llamada recursiva está "más cerca" de alguno de los casos base.
- Pasos a seguir:
  1. Definir el tipo
  2. Enumerar los casos
  3. Definir los casos bases
  4. Completar la definición
     - Si conozco el resultado para un valor "menor" ¿cómo lo puedo utilizar para resolver el problema?
  5. Generalizar y simplificar

Recursión como técnica de diseño

Aproximación: Divide y vencerás
Para resolver un problema P divídelo en varios problemas más pequeños $P_1 \ldots P_n$ de tal forma que:

1. Resolver $P_1 \ldots P_n$ es más sencillo que resolver P directamente
2. La solución de P se obtiene combinando las soluciones de $P_1 \ldots P_n$

Recursión numérica

Definición recursiva de la función factorial:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{e.o.c.} \end{cases}$$

```
-- Usando if
factorialI :: Int -> Int
factorialI n = if n == 0 then 1 else n * factorialI (n-1)

-- Usando guardas
factorialG :: Int -> Int
factorialG n | n == 0    = 1
             | otherwise = n * factorialG (n-1)

-- Usando ecuaciones
factorialE :: Int -> Int
factorialE 0 = 1
factorialE n = n * factorialE (n-1)
```

Recursión numérica (Traza)

```
*Main> factorialE 3
6
```

Traza del cálculo realizado:

```
factorialE 3 =(2ª ecuación)=> 3 * factorialE (3 - 1)
             =(def. -)     => 3 * factorialE 2
             =(2ª ecuación)=> 3 * (2 * factorialE (2 - 1))
             =(def. -)     => 3 * (2 * factorialE 1)
             =(2ª ecuación)=> 3 * (2 * (1 * factorialE (1-1)))
             =(def. -)     => 3 * (2 * (1 * factorialE 0))
             =(1ª ecuación)=> 3 * (2 * (1 * 1))
             =(def. *)     => 6
```

Recursión numérica

Sucesión de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

$$fib(n) = \begin{cases} n & \text{si } n = 0 \text{ ó } n = 1 \\ fib(n-2) + fib(n-1) & \text{e.o.c.} \end{cases}$$

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

Recursión numérica (Traza)

```
*Main> fibonacci 4
3
```

Traza del cálculo realizado:

```
fibonacci 4
        =(3ª ecuación)=>      fibonacci (4 - 2) + fibonacci (4 - 1)
        =(def. -)     =>      fibonacci 2 + fibonacci 3
        =(3ª ecuación)=>      fibonacci (2 - 2) + fibonacci (2 - 1) +
                               fibonacci (3 - 2) + fibonacci (3 - 1)
        =(def. -)     =>      fibonacci 0 + fibonacci 1 +
                               fibonacci 1 + fibonacci 2
        =(todas)      =>      0 + 1 + 1 + fibonacci (2 - 2) +
                            fibonacci (2 - 1) +
        =(def. + y -) =>      2 + fibonacci 0 + fibonacci 1
        =(1ª y 2ª ecuación)=> 2 + 0 + 1
        =(def. +)     =>      3
```

Recursión numérica

Aproximar el número pi utilizando la suma de Leibnitz:

$$\frac{\pi}{4} = \sum_{k \geq 0} \frac{(-1)^k}{2 \cdot k + 1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

```
aproximaPi :: Integer -> Double
aproximaPi n | even n    = 4 * leibnitzP n
             | otherwise = 4 * leibnitzI n

leibnitzP :: Integer -> Double
leibnitzP 0 = 1
leibnitzP n = leibnitzI (n - 1) + recip (2 * (fromIntegral n) + 1)

leibnitzI :: Integer -> Double
leibnitzI n = leibnitzP (n - 1) - recip (2 * (fromIntegral n) + 1)
```

```
*Main> aproximaPi 10000
3.1416926435905346
```

Recursión sobre listas

Calcular la suma los elementos de una lista numérica.

```
sumaLista :: Num a => [a] -> a
sumaLista []     = 0
sumaLista (x:ls) = x + sumaLista ls
```

Recursión sobre listas (Traza)

```
*Main> sumaLista [3,5,2,8]
18
```

Traza del cálculo realizado:

```
sumaLista [3,5,2,8]
      =(2ª ecuación)          => 3 + sumaLista [5,2,8]
      =(2ª ecuación)          => 3 + 5 + sumaLista [2,8]
      =(def. + y 2ª ecuación)=> 8 + 2 + sumaLista [8]
      =(def. + y 2ª ecuación)=> 10 + 8 + sumaLista []
      =(def. + y 1ª ecuación)=> 18 + 0
      =(def. +)               => 18
```

Recursión sobre listas

Determinar si los elementos de una lista están ordenados de menor a mayor.

```
ordenada :: Ord a => [a] -> Bool
ordenada []          = True
ordenada [x]         = True
ordenada (x:(y:ls)) = x <= y && ordenada (y:ls)
```

```
*Main> ordenada [3,5,2,8]
False
*Main> ordenada "abbccddd"
True
```

Recursión sobre listas

Producto escalar de "vectores": $\vec{x} \cdot \vec{y} = \sum x_i \cdot y_i$

```
productoEscalar :: Num a => [a] -> [a] -> a
productoEscalar [] _          = 0
productoEscalar (x:xs) (y:ys) = x*y + productoEscalar xs ys
```

```
*Main> productoEscalar [1,1,1] [1,2,3]
6
*Main> productoEscalar [1,3,5,7] [8,6,4,2]
60
```

Construcción de listas por recursión

Construir la lista con los naturales hasta n.

```
sucDecreciente :: Int -> [Int]
sucDecreciente 0 = [0]
sucDecreciente n = n:(sucDecreciente (n - 1))
```

Construcción de listas por recursión (Traza)

```
*Main> sucDecreciente 2
[2,1,0]
```

Traza del cálculo realizado:

```
sucDecreciente 2
      =(2ª ecuación)=> 2:(sucDecreciente (2 - 1))
      =(def. -)=>      2:(sucDecreciente 1)
      =(2ª ecuación)=> 2:1:(sucDecreciente (1 - 1))
      =(def. -)=>      2:1:(sucDecreciente 0)
      =(1ª ecuación)=> 2:1:[0]
      =(def. :)=>      [2,1,0]
```

Construcción de listas por recursión

Quitar la primera aparición de un determinado elemento.

```
quitaPrimera :: Eq a => a -> [a] -> [a]
quitaPrimera _ []     = []
quitaPrimera x (y:ls) | x == y    = ls
                      | otherwise = y:(quitaPrimera x ls)
```

```
*Main> quitaPrimera 'a' "cdabae"
"cdbae"
*Main> quitaPrimera 1 [5..10]
[5,6,7,8,9,10]
```

Construcción de listas por recursión

Cambiar las apariciones de un elemento 'a' por otro 'b'.

```
cambia :: Eq a => a -> a -> [a] -> [a]
cambia _ _ []     = []
cambia x y (z:ls) | x == z    = y:(cambia x y ls)
                  | otherwise = z:(cambia x y ls)
```

```
*Main> cambia "tomate" "jamón" ["pan", "lechuga", "tomate", "pan"]
["pan","lechuga","jamón","pan"]
```

Recursión terminal (tail recursion o recursión de cola)

- Hasta ahora hemos visto como la recursión se usa para ir construyendo la solución en cada llamada recursiva.
- Otra forma de hacerlo es con **recursión terminal (o de cola)**, donde la solución está en lo último que se evalúa.
- Suele requerir **recursión con acumulador** (aunque no siempre):
  - Si se necesita propagar un valor de una llamada recursiva a las siguientes
  - Requiere de un argumento extra donde "acumular" la solución parcial en la llamada recursiva.
  - Suele requerir de una función auxiliar donde añadir ese argumento acumulador, y la solución llama a ésta con un valor inicial.

Recursión terminal

```
factorialT :: Int -> Int
factorialT = factorialAux 1

factorialAux :: Int -> Int -> Int
factorialAux ac 0 = ac
factorialAux ac n = factorialAux (n * ac) (n - 1)
```

Recursión terminal (Traza)

```
*Main> factorialT 4
24
```

Traza del cálculo realizado:

```
factorialT 4
      =(def. factorialT)=> factorialAux 1 4
      =(2ª ecuación)=>     factorialAux (4 * 1) (4 - 1)
      =(def. * y -)=>      factorialAux 4 3
      =(2ª ecuación)=>     factorialAux (3 * 4) (3 - 1)
      =(def. * y -)=>      factorialAux 12 2
      =(2ª ecuación)=>     factorialAux (2 * 12) (2 - 1)
      =(def. * y -)=>      factorialAux 24 1
      =(2ª ecuación)=>     factorialAux (1 * 24) (1 - 1)
      =(def. * y -)=>      factorialAux 24 0
      =(1ª ecuación)=>     24
```

Recursión terminal numérica

```
fibonacciT :: Int -> Int
fibonacciT = fibonacciAux 0 1

fibonacciAux :: Int -> Int -> Int -> Int
fibonacciAux ac1 _ 0   = ac1
fibonacciAux _ ac2 1   = ac2
fibonacciAux ac1 ac2 n = fibonacciAux ac2 (ac1 + ac2) (n-1)
```

```
*Main> fibonacciT 7
13
```

Recursión terminal (Traza)

```
*Main> fibonacciT 4
3
```

Traza del cálculo realizado:

```
fibonacciT 4
      =(def. fibonacciT)=> fibonacciAux 0 1 4
      =(3ª ecuación)=>     fibonacciAux 1 (0 + 1) (4 - 1)
      =(def. + y -)=>      fibonacciAux 1 1 3
      =(3ª ecuación)=>     fibonacciAux 1 (1 + 1) (3 - 1)
      =(def. + y -)=>      fibonacciAux 1 2 2
      =(3ª ecuación)=>     fibonacciAux 2 (1 + 2) (2 - 1)
      =(def. + y -)=>      fibonacciAux 2 3 1
      =(2ª ecuación)=>     3
```

Recursión terminal numérica

Determinar si un número natural es un cuadrado perfecto, es decir, es el cuadrado de algún otro número natural.

```
esCuadradoPerfecto :: Int -> Bool
esCuadradoPerfecto n = buscaCuadradoDesde 0 n

buscaCuadradoDesde :: Int -> Int -> Bool
buscaCuadradoDesde m n | m * m == n = True
                       | m * m > n  = False
                       | otherwise  = buscaCuadradoDesde (m+1) n
```

```
*Main> esCuadradoPerfecto 9
True
*Main> esCuadradoPerfecto 15
False
```

Recursión terminal numérica

Determinar si un número natural es primo.

```
esPrimo :: Integral a => a -> Bool
esPrimo 2 = True
esPrimo n | n < 2     = False
          | even n    = False
          | otherwise = n == (menorDivisor 3 n)

menorDivisor :: Integral a => a -> a -> a
menorDivisor k n | k * k > n        = n
                 | rem n k == 0     = k
                 | otherwise        = menorDivisor (k + 2) n
```

```
*Main> esPrimo 7
True
*Main> esPrimo 15
False
*Main> menorDivisor 3 65
5
*Main> menorDivisor 3 53
53
```

Recursión terminal numérica y con listas

```
sumaListaT :: Num a => [a] -> a
sumaListaT = sumaListaAux 0

sumaListaAux :: Num a => a -> [a] -> a
sumaListaAux ac []     = ac
sumaListaAux ac (x:ls) = sumaListaAux (x + ac) ls
```

```
*Main> sumaListaT [3,5,2,8]
18
```

Recursión terminal con listas

```
cambiaT :: Eq a => a -> a -> [a] -> [a]
cambiaT = cambiaAux []

cambiaAux ac _ _ []     = ac
cambiaAux ac x y (z:ls)
  | x == z    = cambiaAux (ac ++ [y]) x y ls
  | otherwise = cambiaAux (ac ++ [z]) x y ls
```

```
*Main> cambiaT "tomate" "jamón" ["pan", "lechuga", "tomate", "pan"]
["pan","lechuga","jamón","pan"]
```

Esquema general sobre listas

En muchas ocasiones las funciones recursivas sobre listas tienen el mismo esquema:

```
op1 :: a -> b -> b
funcion1 :: [a] -> b
funcion1 [] = valor1
funcion1 (x:ls) = op1 x (funcion1 ls)

op2 :: b -> a -> b
funcion2 :: [a] -> b
funcion2 = funcion2Aux valor2

funcion2Aux :: b -> [a] -> b
funcion2Aux ac [] = ac
funcion2Aux ac (x:ls) = funcion2Aux (op2 ac x) ls
```

En el próximo tema veremos cómo las funciones de *plegado* nos permiten una definición más simple.

```
funcion1 = foldr op1 valor1

funcion2 = foldl op2 valor2
```

Comparativa recursión (I)

```
sumaLista :: Num a => [a] -> a
sumaLista []     = 0
sumaLista (x:ls) = x + sumaLista ls

sumaListaT :: Num a => [a] -> a
sumaListaT = sumaListaAux 0
sumaListaAux :: Num a => a -> [a] -> a
sumaListaAux ac []     = ac
sumaListaAux ac (x:ls) = sumaListaAux (ac + x) ls
```

Comparativa recursión (II)

```
cambia :: Eq a => a -> a -> [a] -> [a]
cambia _ _ [] = []
cambia x y (z:ls) = (if x == z  then y else z):(cambia x y ls)

cambiaT :: Eq a => a -> a -> [a] -> [a]
cambiaT x y = cambiaAux x y []
cambiaAux :: Eq a => a -> a -> [a] -> [a] -> [a]
cambiaAux _ _ ac [] = ac
cambiaAux x y ac (z:ls) =
   cambiaAux x y (ac ++ (if x == z then [y] else [z])) ls
```

Comparativa recursión (III)

```
concatenaListas :: [[a]] -> [a]
concatenaListas [] = []
concatenaListas (xs:xss) = xs ++ concatenaListas xss
```

Versión con recursión terminal?

Comparativa recursión (III)

```
concatenaListas :: [[a]] -> [a]
concatenaListas [] = []
concatenaListas (x:ls) = x ++ concatenaListas ls

concatenaListasT :: [[a]] -> [a]
concatenaListasT = concatenaListasAux []
concatenaListasAux ac [] = ac
concatenaListasAux ac (x:ls) = concatenaListasAux (x ++ ac) ls
```

Comparativa recursión (IV)

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece x [] = False
pertenece x (y:ls) = x == y || pertenece x ls
```

Versión con recursión terminal?

Comparativa recursión (IV)

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece x [] = False
pertenece x (y:ls) = x == y || pertenece x ls

perteneceT :: Eq a => a -> [a] -> Bool
perteneceT x = perteneceAux x False
perteneceAux :: Eq a => a -> Bool -> [a] -> Bool
perteneceAux _ ac [] = ac
perteneceAux x ac (y:ls) = perteneceAux x ((x == y) || ac) ls
```

Ejercicio propuesto

Un número n es *k-belga* si la sucesión cuyo primer elemento es k y cuyos elementos se obtienen sumando reiteradamente los dígitos de n contiene a n. Por ejemplo:

- El 18 es 0-belga, porque a partir del 0 vamos a ir sumando sucesivamente 1, 8, 1, 8, ... hasta llegar o sobrepasar el 18: 0, 1, 9, 10, 18, ... Como se alcanza el 18, resulta que el 18 es 0-belga.
- El 19 no es 1-belga, porque a partir del 1 vamos a ir sumando sucesivamente 1, 9, 1, 9, ... hasta llegar o sobrepasar el 19: 1, 2, 11, 12, 21, ... Como no se alcanza el 19, resulta que el 19 no es 1-belga.

Ejercicio propuesto

Definir la función

```
esBelga :: Integer -> Integer -> Bool
```

tal que (esBelga k n) se verifica si n es k-belga. Por ejemplo,

```
esBelga 0 18                                  ==  True
esBelga 1 19                                  ==  False
esBelga 0 2016                                ==  True
[x | x <- [0..30], esBelga 7 x]               ==  [7,10,11,21,27,29]
[x | x <- [0..30], esBelga 10 x]              ==  [10,11,20,21,22,24,26]
length [n | n <- [1..9000], esBelga 0 n]      ==  2857
```

Bibliografía

R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
Capítulo 3: Números
Capítulo 4: Listas

G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
Chapter 6: Recursive functions

B. O'Sullivan, D. Stewart y J. Goerzen. *Real World Haskell*. O'Reilly, 2008.
Chapter 2: Types and Functions

B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. Razonando con Haskell. Thompson, 2004..
Capítulo 2: Introducción a Haskell
Capítulo 6: Programación con listas

S. Thompson. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999.
Chapter 4: Designing and writing programs