



28-10-2024

# PRÁCTICA 2

## Modelos avanzados de computación

*PRIMEROS PASOS CON FUNCIONES EN HASKELL*



wadadi sidelgaum limam.

## Contenido

1. Introducción.....	3
2. Ejercicios .....	3
2.1. Combinatoria de N sobre K .....	3
2.2. Cálculo de Raíces .....	5
2.3. Calculo de Intersección.....	6
2.4. Comprobar si pertenece o no.....	7

## 1. Introducción

EL objetivo de esta práctica es aprende de utilizar diferentes métodos de resolución para resolver algunos problemas, a la hora de definir funciones existen diferentes posibilidades:

- Utilizando varias ecuaciones: escribiendo cada ecuación que actúa de forma distinta en una línea.
- Utilizando Guardas: nos permite definir las condiciones de forma legible.
- Sentencia IF THEN ELSE : nos ayuda para evaluar casos específicos.
- Con sentencia case: sirve para controlar distintos patrones a partir de una expresion.
- Definiciones locales (where): Permiten definir funciones o variables dentro de una función para evitar redundancia.

Hay que resolver cad uno de los siguientes ejercicios utilizando las posibilidades mencionada anteriormente:

1. Comb n k : Calculo de combinaciones de n elementos escogidas de K en K.
2. raices: obtener las raices siguiendo la ecuacion de segundo grado.
3. Interseccion: calcular los elementos comunes entre dos listas.
4. Pertence: comprobar que un elemento pertence a una lista.

## 2. Ejercicios

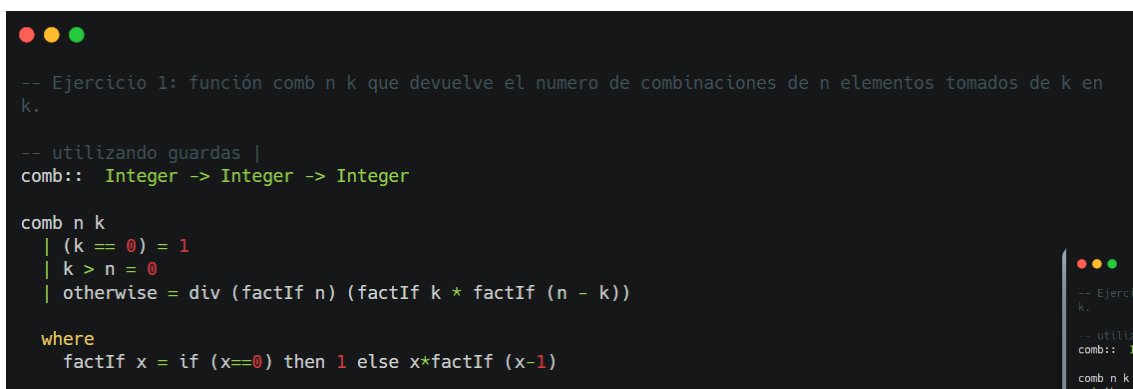
### 2.1. Combinatoria de N sobre K

En este ejercicio tenemos que calcular el número de combinaciones posibles de n elementos tomados de k en k, utilizando la fórmula:

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

#### Soluciones:

1. Guardas: mediante las cláusulas definimos los casos en lo que no es necesario el uso de la función, en primer lugar, si k es igual a 0 el resultado serio 1, ya que la factorial de 0 es 1 y nos queda factorial de n entre factorial de n, después tenemos que si k es mayor que n la factorial de un numero negativo no existe, en cualquier otro caso se calcula los datos con la formula mediante una función auxiliar.



```
-- Ejercicio 1: función comb n k que devuelve el numero de combinaciones de n elementos tomados de k en k.
-- utilizando guardas |
comb :: Integer -> Integer -> Integer

comb n k
  | (k == 0) = 1
  | k > n = 0
  | otherwise = div (factIf n) (factIf k * factIf (n - k))

where
  factIf x = if (x==0) then 1 else x*factIf (x-1)
```

Figura 1: Código del ejercicio comb con guardas

2. Implementamos los mismos casos anteriores con la sentencia if then else.

```
-- Ejercicio 1: función comb n k que devuelve el numero de combinaciones de n elementos tomados de k en k.

-- utilizando guardas IF THEN ELSE
combIf:: Integer -> Integer -> Integer

combIf n k = if (k == 0) then 1
              else if (k>n) then 0
              else div (factIf n) (factIf k * factIf (n - k))

where
  factIf x = if (x==0) then 1 else x*factIf (x-1)
```

Figura 2: Código del ejercicio comb con if then else

3. Implementamos la misma función con la sentencia case, donde comprobamos si el valor de k, si es 0 se devuelve 1 en caso contrario devuelve el resultado al aplicar la formula.

```
-- Ejercicio 1: función comb n k que devuelve el numero de combinaciones de n elementos tomados de k en k.

-- utilizando case
combCase:: Integer -> Integer -> Integer

combCase n k = case (k) of
  0 -> 1
  _ -> div (factIf n) (factIf k * factIf (n - k))

where
  factIf x = if (x==0) then 1 else x*factIf (x-1)
```

Figura 3: Código del ejercicio comb con case

## 2.2. Cálculo de Raíces

En este ejercicio tenemos que hacer el cálculo de las raíces de una ecuación cuadrática de la forma  $ax^2+bx+c=0$ , siguiendo la siguiente ecuación:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

### Soluciones:

1. Guardas: definimos los casos que hacen que la formula no se pueda calcular, en primer lugar, comprobamos que el valor a no es cero ya que una división por cero atiende a infinito, otro caso seria si el valor de b cuadrado es menor que el cuatro por el valor a y por el valor c, entonces se devuelve lista vacía como en el caso anterior, finalmente se calcula las valores de las raíces con la formula mediante una función auxiliar definida por la cláusula where.

```
-- Ejercicio 2: función raices a b c
-- utilizando ||
raices_gua a b c
  | a==0 = []
  | ((b*b)-(4*a*c))<0 = []
  | otherwise = aux a b c
  where
    aux a b c = [(-b + sqrt ((b*b) - (4 *a*c)))/(2*a),(-b - sqrt ((b*b) - (4*a*c)))/(2*a)]
```

Figura 4: Código del ejercicio raíces con guardas

2. Implementación con la sentencia if then else.

```
-- Ejercicio 2: función raices a b c
-- utilizando if then else
raices a b c = if (a==0) then []
               else if ((b*b)<(4*a*c)) then []
               else aux a b c
  where
    aux a b c = [(-b + sqrt ((b*b) - (4 *a*c)))/(2*a),(-b - sqrt ((b*b) - (4*a*c)))/(2*a)]
```

Figura 5: Código del ejercicio raíces con if then else

## 2.3. Calculo de Intersección

Este ejercicio consiste en devolver los números que pertenecen entre dos listas, donde cada lista tiene dos valores que define intervalos de números entre ellos.

### Soluciones:

1. Guardas: primero comprobamos los casos bases donde vemos si hay algún elemento coincide con las dos listas, o sea, si el valor máximo de la lista 1(segundo elemento) es menor que el valor mínimo de la lista 2(primer elemento), o viceversa se devuelve la lista vacía, en case contrario se devuelve una lista que contiene dos valores que definen los intervalos de la intersección de las dos listas, donde el primero es el máximo entre el primer elemento de la lista 1 y de la lista 2 y el segundo es el mínimo entre el segundo elemento de cada lista.

```
-- Ejercicio 3: define la función intersección, tal que (interseccion i1 i2) es la intersección de los
intervalos i1 e-- i2

-- utilizando guardas |
--interseccion :: Ord a => [a] -> [a] -> [a]
interseccion [x, y] [w, z]
  | y < w = []
  | x > z = []
  | otherwise = [(max x w), (min y z)]    -- [(max x w)..(min y z)]
```

Figura 6: Código del ejercicio interseccion con guardas

2. Implementación con la sentencia if then else, donde interpretamos los casos anteriores cambiando la sintaxis.

```
-- Ejercicio 3: define la función intersección, tal que (interseccion i1 i2) es la intersección de los
intervalos i1 e-- i2

-- utilizando if then else
--interseccionIF :: Ord a => [a] -> [a] -> [a]
interseccionIF [x, y] [w, z] = if (y < w) then [] else if (x > z) then []
                               else [(max x w), (min y z)] |
```

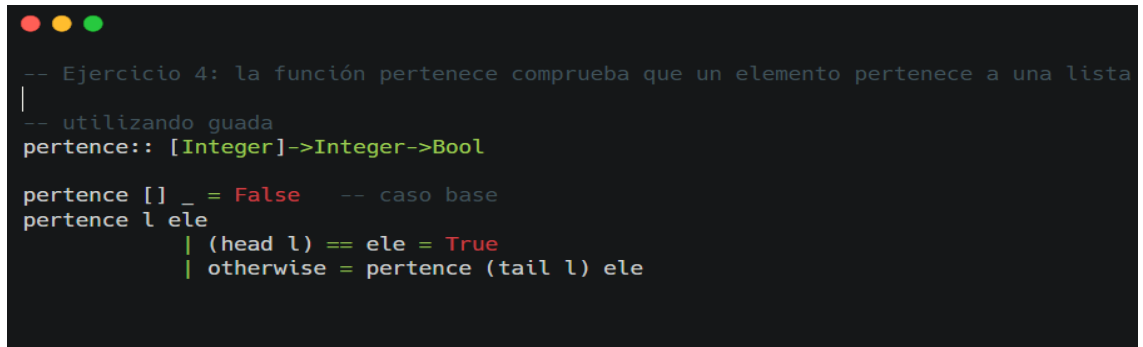
Figura 7: Código del ejercicio interseccion con if then else

## 2.4. Comprobar si pertenece o no

Este ejercicio consiste en comprobar si un elemento pertenece a una lista, el elemento puede ser un valor como veremos en el siguiente apartado o tuplas como veremos al final de esta sección.

### Soluciones:

1. Guardas: en esta función tenemos dos casos, el caso base donde si la lista es vacía devolvemos un falso, y el otro caso comprobamos si el elemento que se encuentra de la cabecera de la lista es igual que el valor recibido por parámetro, si lo es devuelve verdadero, en caso contrario se hace la llamada recursiva con el resto de la lista sin la cabecera, de manera que si el elemento no pertenece a la lista la llamada recursiva llega finalmente al caso base y devuelve falso.

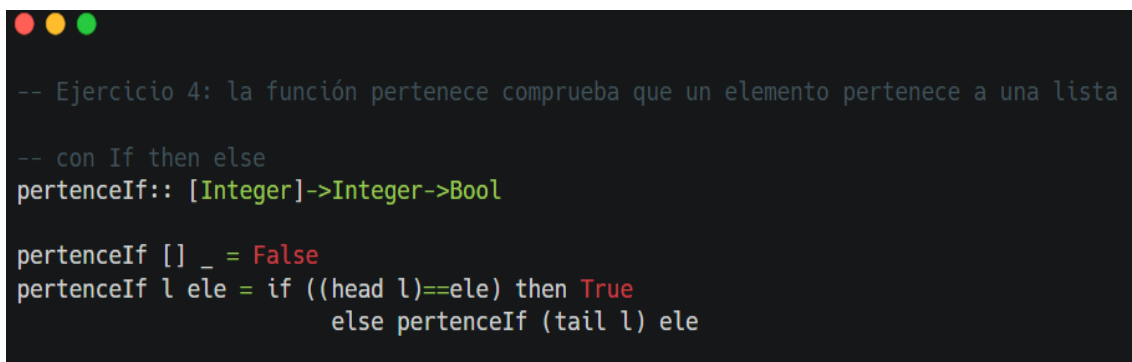


```
-- Ejercicio 4: la función pertenece comprueba que un elemento pertenece a una lista
-- utilizando guada
pertenece:: [Integer]->Integer->Bool

pertenece [] _ = False    -- caso base
pertenece l ele
    | (head l) == ele = True
    | otherwise = pertenece (tail l) ele
```

Figura 8: Código del ejercicio pertenece con guardas

2. Implementación con la sentencia if then else.

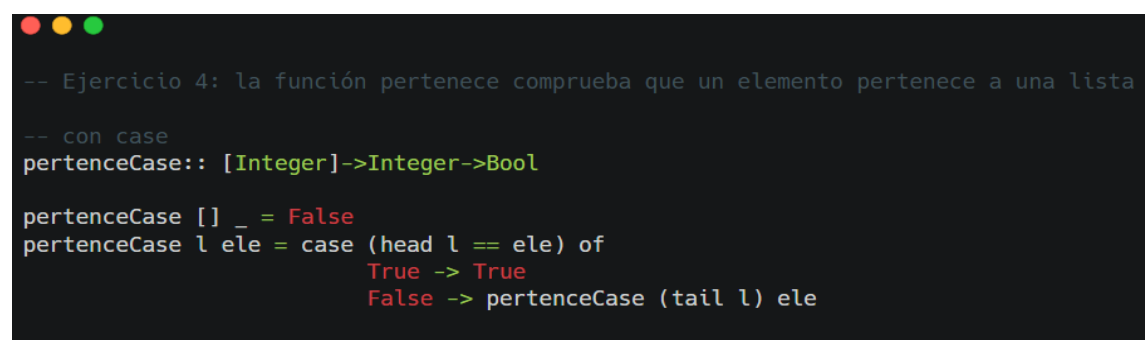


```
-- Ejercicio 4: la función pertenece comprueba que un elemento pertenece a una lista
-- con If then else
perteneceIf:: [Integer]->Integer->Bool

perteneceIf [] _ = False
perteneceIf l ele = if ((head l)==ele) then True
                    else perteneceIf (tail l) ele
```

Figura 9: Código del ejercicio pertenece con if then else

3. Implementación con la cláusula case.

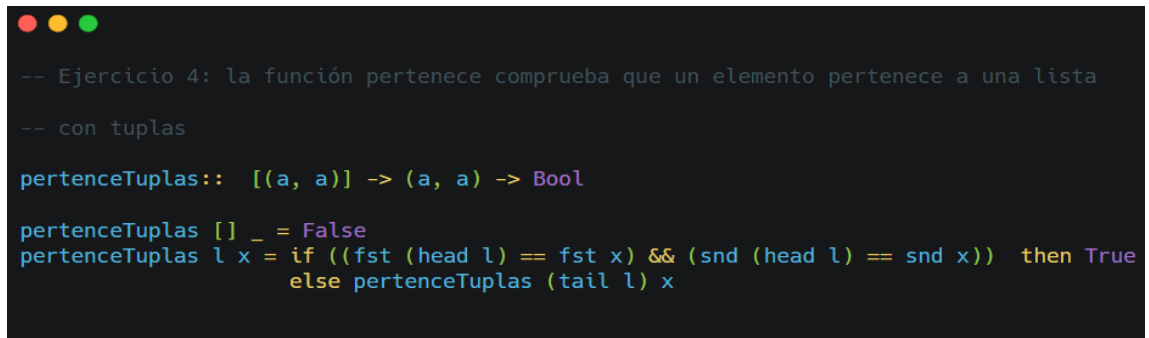


```
-- Ejercicio 4: la función pertenece comprueba que un elemento pertenece a una lista
-- con case
perteneceCase:: [Integer]->Integer->Bool

perteneceCase [] _ = False
perteneceCase l ele = case (head l == ele) of
    True -> True
    False -> perteneceCase (tail l) ele
```

Figura 10: Código del ejercicio pertenece con case

4. Implementación con tuplas: se hace uso de los mismos casos anteriores, pero al ser los elementos tuplas para comprobar si dos tuplas son iguales habría que comprobar el primero de una con el primero y el segunda de una tupla con la otra.

A screenshot of a code editor with a dark background and syntax-highlighted Haskell code. The code defines a function 'perteneceTuplas' that checks if an element 'x' is in a list 'l' of tuples. The function signature is 'perteneceTuplas :: [(a, a)] -> (a, a) -> Bool'. The implementation uses a recursive pattern: it returns 'False' for an empty list, and for a non-empty list, it checks if the first element's first component matches 'x' and its second component is also 'x'. If not, it recursively calls 'perteneceTuplas' on the tail of the list.

```
-- Ejercicio 4: la función pertenece comprueba que un elemento pertenece a una lista
-- con tuplas

perteneceTuplas :: [(a, a)] -> (a, a) -> Bool

perteneceTuplas [] _ = False
perteneceTuplas l x = if ((fst (head l) == fst x) && (snd (head l) == snd x)) then True
                        else perteneceTuplas (tail l) x
```

Figura 11: Código del ejercicio pertenece con tuplas