SESION 5

INTRODUCCIÓN WINHUGS

Universidad de Huelva

5.1. REPASO

Formas de definir una función

Ecuaciones

Guardas

If then else

Case

y definiciones locales

Formas de definir una función: pertenece

Ecuaciones

Guardas

If then else

Case

y definiciones locales

```
pertenece :: [Integer] -> Integer -> Bool
pertenece [] _ = False
pertenece (x:xs) elemento = x == elemento || pertenece xs elemento
```

Formas de definir una función: pertenece

Ecuaciones

Guardas

If then else

Case

y definiciones locales

```
pertenece:: [Integer]->Integer->Bool
pertenece [] _ = False    -- caso base
pertenece lista elemento
    | (head lista) == elemento = True
    | otherwise = pertenece (tail lista) elemento
```

Formas de definir una función: pertenece

Ecuaciones

Guardas

If then else

Case

y definiciones locales

```haskell
perteneceIf:: [Integer]->Integer->Bool
perteneceIf [] _ = False
perteneceIf lista elemento =
    if ((head lista)==elemento) then True
    else perteneceIf (tail lista) elemento
```

Formas de definir una función: pertenece

Ecuaciones

Guardas

If then else

Case

y definiciones locales

```
perteneceCase:: [Integer]->Integer->Bool
perteneceCase [] _ = False
perteneceCase lista elemento =
    case (head lista == elemento) of
        True -> True
        False -> perteneceCase (tail lista) elemento
```

Formas de definir una función: pertenece

Ecuaciones

Guardas

If then else

Case (tuplas)

y definiciones locales

```haskell
perteneceLaTupla :: Eq a => [(a, a)] -> (a, a) -> Bool
perteneceLaTupla [] _ = False
perteneceLaTupla lista tuple  =
    if ((fst (head lista) == fst tuple) && (snd (head lista) == snd tuple)) then True
    else perteneceLaTupla (tail lista) tuple
```

Formas de definir una función: pertenece

Ecuaciones

Guardas

If then else

Case

y definiciones locales

```haskell
pertenceLaTupla :: Eq a => [(a, a)] -> (a, a) -> Bool
pertenceLaTupla [] _ = False
pertenceLaTupla (cabeza:resto) elemento
     | esIgual cabeza elemento = True
     | otherwise = pertenceLaTupla resto elemento
  where
    esIgual (a1, a2) (b1, b2) = (a1 == b1) && (a2 == b2)
```

Índice

¿Qué vamos a ver?

Instalación de ghc (Glasgow Haskell Compiler)

Concepto de Currificación

Concepto de principio de inducción y recursividad

Listas intensionales

Ejercicios

Currificación de funciones: concepto de currificación

La currificación: cualquier función de múltiples argumentos puede ser expresada como una *secuencia de funciones de un solo argumento.*

Proviene del lógico Haskell Curry, quien demostró que la función **(a, b) -> c** puede representarse como realmente como **a -> (b -> c).**

Por tanto, cualquier función que normalmente toma múltiples parámetros se convierte en una sucesión de funciones que toman un parámetro y devuelven otra función.

Currificación de funciones… en haskell

Consiste en realizar una llamada a una función con los parámetros que están más a la izquierda.

Una función puede recibir un número variable de argumentos, dando valores de forma parcial de izquierda a derecha.

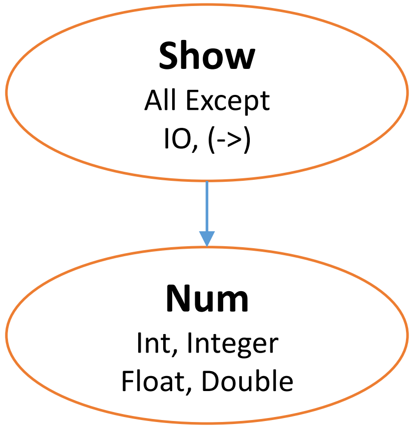
Si la función tiene 3 argumentos puedo dar el primero y dejar los otros dos… (o generarlos al vuelo)

También puedo poner los dos primero fijos y dar el último

```
Main> map (suma 1 1) [1,2,3]
[3,4,5] :: [Int]
```

Currificación de funciones: suma x y z = x + y + z

```
Main> map (suma 2 4) [1..4]
[7,8,9,10]
Main> map (suma 2) [1..4]
ERROR - Cannot find "show" function for:
*** Expression : map (suma 2) (enumFromTo 1 4)
*** Of type    : [Int -> Int]
```



¿Qué devuelve? ¿por qué da el error? ¿es un error?

Devuelve una lista de funciones y **no se puede mostrar por pantalla -> [(+3),(+4),(+5), (+6)], lista de funciones**

`(a->b) -> (a->a->a). Función parcialmente aplicada.`

Currificación de funciones: suma x y z = x + y + z

Aunque parezca que tiene 3 argumentos, en Haskell realmente de define usando Currificación implícita, lo cual implica que, en vez de tomar los 3 argumentos de una vez, recibe un solo argumento y devuelve una función que espera el siguiente argumento, y así sucesivamente, de izquierda a derecha.

La función se descompone en una serie de funciones anidadas.

suma :: Int -> Int -> Int -> Int	Int ->(Int -> (Int -> Int))
suma x y z = x + y + z	(x+(y+(z)))

Currificación de funciones: suma x y z = x + y + z

La función suma x y z = x + y + z en realidad se interpreta como:

suma x = \y -> (\z -> x + y + z)

Es decir, suma toma un solo argumento x y devuelve otra función \y -> (\z -> x + y + z) que espera el siguiente argumento

Si llamamos a suma con solo el primer argumento, devuelve:

\y -> (\z -> 1 + y + z)   1 + (int->int->int)

Función que espera los argumentos Y y Z.

Currificación de funciones: suma x y z = x + y + z

Ahora aplicamos el segundo argumento usando la función que obtuvimos

(suma 1) +2

Que nos devuelve la siguiente función

\z -> 1 + 2 + z 1+2+(**int**->int)

Esta función fija los valores 1 y 2 y espera el último argumento z para completar la suma

((suma 1) +2) +3

Esto calcula el resultado de 1 + 2 + 3, que es 6

Currificación de funciones: suma x y z = x + y + z

1. sumaTres = \x -> (\y -> (\z -> x + y + z))
2. sumaTres 1 -- Produce la función \y -> (\z -> 2 + y + z)
3. (sumaTres 1) 2 -- Produce la función \z -> 2 + 3 + z
4. ((sumaTres 1) 2) 3 -- Produce el valor 6

Currificación de funciones con tuplas

suma :: Num a => a -> a -> a

suma x y = x + y

map (+ 1) [1..5]

```
Main> map (+ 1) [1..5]
[2,3,4,5,6]
```

¿Qué ocurre si la definimos con una tupla como argumento?

suma2 :: Num a => (a,a) -> a

suma2 (x, y) = x + y

map (suma2 1) [1..5]

```
Main> map (suma2(1)) [1..5]
ERROR - Cannot infer instance
*** Instance   : Num (a -> b)
*** Expression : map (suma2 1) (enumFromTo 1 5)
```

No se puede utilizar de forma parcial

Currificación de funciones con tuplas : ¿Cómo lo resolvemos?

```
curry :: ((a,b) -> c) -> a -> b -> c

Main> map (curry suma2(1)) [1,2,4]

[2,3,5]
```

¿Y si queremos definir la suma currificada?

```
sumaCurrificada      :: Num a => a -> a -> a

sumaCurrificada x = (\y -> x + y)
```

Currificación de funciones: ¿Cómo lo resolvemos?

curry :: ((a,b) -> c) -> a -> b -> c

La función **curry** en Haskell convierte una función que toma una tupla como argumento en una función curried o currificada (es decir, en una función que toma sus argumentos uno a uno), transforma una función que espera una tupla en una que recibe sus elementos como argumentos separados.

- **((a, b) -> c)** es una función que toma una tupla (a, b) y devuelve un valor de tipo c.
- curry convierte esta función en una función curried de **tipo a -> b -> c**, es decir, una función que toma dos argumentos individuales (de tipo a y b) y devuelve un valor de tipo c.

Currificación de funciones: ¿Cómo lo resolvemos?

curry :: ((a,b) -> c) -> a -> b -> c

Ejemplo con la multiplicacion:

multiplicar :: (Int, Int) -> Int

multiplicar (x, y) = x * y

multiplicarCurried :: Int -> Int -> Int

multiplicarCurried = curry multiplicar

```
Main> multiplicarCurried 5 5
25
Main> multiplicar 5 5
ERROR - Type error in application
*** Expression     : multiplicar 5 5
*** Term           : multiplicar
*** Type           : (Int,Int) -> Int
*** Does not match : a -> b -> c

Main> multiplicar (5,5)
25
Main>
```

Currificación de funciones: usado con lista de funciones

Veamos un ejemplo

```
Main> miFuncion [(+1), (+2), (+3)] [1..3]
[2,4,6]

¿y si dividimos y restamos?

Main> miFuncion [(/4), (**4), (3-)] [1..3]
[0.25,16.0,0.0]
Main> miFuncion [(div 4), (^4), (3-)] [1..4]
[4,16,0]
```

Currificación de funciones: lista de funciones

Veamos un ejemplo. ¿Está completa?

```
miFuncion::[(a->b)] -> [a] -> [b]
miFuncion (f:fs) (x:xs) = (f x):(miFuncion fs xs)
```

```
miFuncion::[(a->b)] -> [a] -> [b]
miFuncion [] _ = []
miFuncion (f:fs) (x:xs) = (f x):(miFuncion fs xs)
```

Hay que añadir el caso base

Currificación de funciones: función map aplicando currifucación

```
Main> funcionMapCurrificada (map (suma3 3) [1,2,3]) [1,2,3]
[5,7,9] :: [Int]
```

¿Cuál sería su implementación? 5 minutos

```haskell
suma3 ::  Int -> Int-> Int -> Int
suma3 x y z = x + y + z

funcionMapCurrificada :: [a -> b] -> [a] -> [b]
funcionMapCurrificada [] _ = []
funcionMapCurrificada (f:fs) (x:xs) = f x : (funcionMapCurrificada fs xs)
```

Principio de inducción

Sea x que pertenece al conjunto S, **S tiene que ser ordenable**

Queremos probar si la propiedad P es cierta para todo elemento del conjunto S

1) P es cierta para el n0 (el elemento mas pequeño)

Ejemplo 1: naturales 1, 2, 3, 4 …

Ejemplo 2: listas [], [_],[_,_], …

Ejemplo 3: arboles binarios: nil, a(_, nil, nil), …

2) **Si P es cierta para n-1 entonces puedo afirmar que puede ser cierta para n: P(n-1) -> P(n)**

Principio de inducción

Suma todos los elementos de una lista aplicando el principio de inducción matemático.

1) P(n0)

2) P(n-1) -> P(n)

3) ¿n-1 en listas? Operador ":", que separa la cabeza del resto.

4) Suma es cierto si devuelve la suma de los elementos de la lista que se pasa por parámetros como arg. 1

suma :: Num a => [a] -> a

suma [] = 0 | suma (cabeza:resto) = cabeza + (suma resto). Pensemos la función como algo estático.

Principio de inducción

Suma todos los elementos de una lista aplicando el principio de inducción matemático.

1)	suma [1,2,3] ->	suma [2, 3]	-> suma [3]	-> suma []
2)	1+5	2+3	3+0	0

Y si quisiéramos demostrar los siguiente, ¿Cuándo sería cierto?

```
miFuncionLista::[(a->b)] -> [a] -> [[b]]
```

```
Ejemplo: map (+1) (+2) (+3) [1,2,3] -> [[2,3,4],[3,4,5][4,5,6]]
```

Principio de inducción

miFuncionLista es cierto si devuelve una lista de listas, resultado de aplicar las funciones de la lista de funciones del primer argumento a todos los valores de la lista de valores del segundo argumento.

```
miFuncionLista::[(a->b)] -> [a] -> [[b]]
miFuncionLista [] _ = []
miFuncionLista (f:fs) lista = (map f lista) : (miFuncionLista fs lista)
```

```
Main> miFuncionLista [(+2), (+3)] [1,2,3]
[[3,4,5],[4,5,6]] :: [[Integer]]
```

Principio de inducción: definimos la función map de otra forma

```
miFuncion2::[(a->b)] -> [a] -> [[b]]
miFuncion2 [] _ = []
miFuncion2 (f:fs) x = miFuncion3 f x : miFuncion2 fs x
```

¿Como sería miFuncion3?

```
Main> miFuncion2 [(+2), (+3)] [1,2,3]
[[3,4,5],[4,5,6]] :: [[Integer]]
```

5.3. LISTAS INTENSIONALES

Listas intensionales

Haskell incluye una sintaxis especial para representar de forma más compacta operaciones sobre listas.

En matemáticas existe una forma “intensiva de definir conjuntos”

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$$

Modificador o función de salida

pertenencia

restricciones

Esto significa que el conjunto contiene todos los dobles de los número naturales que cumplen el predicado

5.3.1. Concepto

Listas intensionales

En haskell las listas intensionales son similares a los conjuntos definidos intensionalmente.

¿Cuáles serían las partes de esta definición intensional?

```
[(x,True) | x <- [1..20]], even x, x < 15]
```

Transformación

Generador

Restricciones

¿Y si quisiéramos todos los números del 50 al 100 cuyo resto al dividir por 7 fuera 3?

```
[ x | x <- [50..100], x `mod` 7 == 3]
```

Listas intensionales

Son muy utilizadas en haskell, ya que permite generar una lista sin tener que utilizar una función con recursividad.

Veamos algunos ejemplos

```
Main> [(1,2*x)|x<-[1..5]]
[(1,2),(1,4),(1,6),(1,8),(1,10)] :: [(Integer,Integer)]
Main> [x==1|x<-[1..5]]
[True,False,False,False,False] :: [Bool]
Main> [f 1|f<-[(+1), (+2), (+3)]]
[2,3,4] :: [Integer]
```

Listas intensionales

Puedo utilizar más de un generador.

Se fija un generador y se aplica a todos los del segundo.

```
Main> [f x | f<-[(==1), (==2), (==3)],x <- [3,4,5]]
[False,False,False,False,False,False,True,False,False] :: [Bool]
Main> [f x | x <- [3,4,5], f<-[(==1), (==2), (==3)]]
[False,False,True,False,False,False,False,False,False] :: [Bool]
```

Listas intensionales

Implementar la función filter2: filter2 :: (a -> Bool) -> [a] -> [a]

```haskell
filter2 :: (a -> Bool) -> [a] -> [a]
filter2 p xs = [x | x <- xs, p x]
```

```
Main> filter2 (==1) [1,2,3]
[1] :: [Integer]
```

Listas intensionales

Implementar sumaCuadradosPares xs es la suma de los cuadrados de los números pares de la lista xs**.**

Por ejemplo: sumaCuadradosPares [1..5] == 20

```haskell
sumaCuadradosPAres :: [Int] -> Int
sumaCuadradosPAres xs = sum [x^2 | x <- xs, even x]
```

```
Main> sumaCuadradosPares [1..5]
20 :: Int
```

Listas intensionales

Generar cada una de las siguientes listas utilizando la notación extendida de listas, con la lista [1..10] como generador. Es decir, cada solución debe tener la siguiente forma, donde se deben completar los blancos y sólo los blancos.

```
[ ____________________ | x <- [1 .. 10] _____________________]
```

De forma más explícita, la respuesta debe utilizar el generador x<- [1.. 10], y no debe añadir a esta definición ninguna llamada a función: por ejemplo, no utilizar reverse [x | x <- [1 .. 10]] para crear la lista [10,9,8,7,6,5,4,3,2,1]. De la misma forma, modificaciones del tipo [x|x <- [10,9..1]] y [x|x <- reverse[1 ..10]] también están prohibidas.

5.3.2.3. Ejercicios prácticos

Listas intensionales: subir a moodle

```
1. [11,12,13,14,15,16,17,18,19,20]
2. [[2],[4],[6],[8],[10]]
3. [[10],[9],[8],[7],[6],[5],[4],[3],[2],[1]]
4. [True,False,True,False,True, False,True,False,True,False]
5. [(3,True),(6,True),(9,True),(12,False),(15,False), (18,False)]
6. [(5,False),(10,True),(15,False),(40,False)]
7. [(11,12),(13,14),(15,16),(17,18),(19,20)]
8. [[5,6,7],[5,6,7,8,9],[5,6,7,8,9,10, 11],[5,6,7,8,9,10,11,12,13]]
9. [21,16,11,6,1]
10. [[4],[6,4],[8,6,4],[10,8,6,4],[12,10,8,6,4]]
```