PRÁCTICA 5

INTRODUCCIÓN WINHUGS

Universidad de Huelva

5.1 REPASO

5.1. Repaso

Recordemos

Tuplas

2-tuplas / 3-tuplas

[(1,2),(8,11),(4,5)] [(1,2,8),(8,11,5),(4,5,0)] [(0,False),(1,True)]

Funciones sobre tuplas

fst, snd, splitAt, span, break, zip, zip3, zipWith, unzip, unzip3

5.1. Repaso

Orden de prioridad

La notación **infix** indica que el operador es infijo, es decir, que se escribe entre los operandos.

En el caso de encadenar operadores no se define ninguna prioridad. (1 == 1 == 1)

La notación **Infixl** indica que, en caso de igualdad de precedencia se evaluará primero la izquierda.

La notación **Infixr** indica que, en caso de igualdad de precedencia se evaluará primero el operador que está más a la derecha.

El operador que mayor precedencia tiene es la composición de funciones “.”.

5.1. Repaso

Notación	Prioridad	Operador
infixr	9	.
infixl	9	!!
infixr	8	^, ^^, **
infixl	7	*, /, `quot`, `rem`, `div`, `mod`
infixl	6	+, -
infixr	5	:
infixr	5	++
infix	4	==, /=, <, <=, >=, >,`elem`,`notElem`
infixr	3	&&
infixr	2	||
infixl	1	>>, >>=
infixr	1	=<<
infixr	0	$, $!, `seq`

5.1. Repaso

Evaluación perezosa

```haskell
soloPrimero::a->b->a
soloPrimero x _ = x
```

Notación (x:xs) -> cabecera:resto

```haskell
suma :: [Int] -> Int
suma [] = 0
suma (x:xs) = x + sum xs
```

5.2 DEFINICIÓN DE FUNCIONES

5.2. Definición de funciones

Formas de definir una función

Tenemos diferentes posibilidades a la hora de definir funciones:

- Utilizando varias ecuaciones (escribiendo cada ecuación en una línea)
- Guardas (en inglés “guards”, barreras, defensas, “|”)
- If then else
- Case
- En definiciones locales

5.2.1 Varias ecuaciones

Formas de definir una función: varias ecuaciones

Factorial de un número entero:

```haskell

{- FACTORIAL VARIAS ECUACIONES -}

factorial_ecuaciones::Int->Int
factorial_ecuaciones 0 = 1
factorial_ecuaciones n = n * factorial_ecuaciones (n-1)

```

¿Existen diferencias? ¿Cuál?

```haskell
{- FACTORIAL VARIAS ECUACIONES -}

factorial_ecuaciones2::Integral a => a-> a
factorial_ecuaciones2 0 = 1
factorial_ecuaciones2 n = n * factorial_ecuaciones2 (n-1)
```

5.2.2. Guardas "|"

Formas de definir una función: utilizando guardas

Factorial de un número entero:

```haskell
{- FACTORIAL GUARDAS -}

--factorial_guarda ::Integer->Integer
--factorial_guarda :: (Num a, Ord a) => a -> a
factorial_guarda :: (Num a, Ord a) => a -> a
factorial_guarda n
  | n ==0    = 1
  | n > 0    = n * factorial_guarda(n-1)
  | otherwise = error "valor negativo"
```

5.2.2. Guardas "|"

Formas de definir una función: utilizando guardas

Comprobar signo de un número:

```
24
25    --positivo :: (Num a, Ord a) => a -> Bool
26    positivo x
27          | x > 0 = True
28          | x < 0 = False
29          | otherwise = True
```

5.2.3. IF THEN ELSE

Formas de definir una función: if then else

Factorial de un número entero / posicionar en intervalo:

```
30
31   {- FACTORIAL IF THEN ELSE -}
32
33   --factorialIF :: Num a => a -> a
34   factorialIF n = if (n==0) then 1 else n*factorialIF (n-1)
35
```

5.2.3. IF THEN ELSE

Formas de definir una función: if then else

Factorial de un número entero / posicionar en intervalo:

```
{- INTERVALO IF THEN ELSE -}

--intervalo :: (Ord a, Num a) => a -> [Char]
intervalo x =
    if x > 75 then "Intervalo 4"
    else if 50 < x && x <= 75 then "Intervalo 3"
    else if 25 < x && x <= 50  then "Intervalo 2"
    else "Primer intervalo"
```

5.2.4. CASE

Formas de definir una función: CASE

case **expresion** of patron -> resultado

patron -> resultado

patron -> resultado

...

La expresión se compara con los patrones. Se utiliza el primer patrón que coincide con la expresión. Si no cubre toda la expresión del caso y no se encuentra un patrón adecuado, se produce un error de tiempo de ejecución.

5.2.4. CASE

Formas de definir una función: CASE

Comprobar si un número es par

```
58
59    -- comprobar si numero par
60    case_par :: Integral a => a -> Bool
61    case_par n = case (mod n 2 == 0) of
62                      True->True
63                      False->False
```

5.2.4. CASE

Formas de definir una función: CASE

Mini calculadora: sumar o restar en función de un parametro

```
65   -- operar segun parametro
66   sumar :: Num a => a -> a -> a
67   sumar x y = x + y
68   restar :: Num a => a -> a -> a
69   restar x y = x - y
70   operar :: (Num a, Num b) => b -> a -> a -> a
71   operar x = case x of
72       1 -> sumar
73       2 -> restar
```

5.3.4. CASE

Formas de definir una función: CASE

Mostrar por pantalla si una lista es vacia, de 1 elemento, 2 elementos o mas

```haskell
-- comprobar tipo de lista
tipolista :: [a] -> String
tipolista xs = "Resultado: " ++ case xs of []       -> "Lista vacia."
                                           (x:[])   -> "Lista con un elemento."
                                           (x:y:[]) -> "Lista de dos elementos."
                                           (x:y:_)  -> "Lista con mas de dos elementos"
```

5.2.5. Definiciones locales

Formas de definir una función: definiciones locales

Es posible definir una función en cualquier punto de otra función.

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

```
85    --divisible a y b
86    divisible::Int->Int->Bool
87    divisible x y = resto == 0
88        where resto = mod x y
```

5.2.5. Definiciones locales

Formas de definir una función: definiciones locales

Es posible definir una función en cualquier punto de otra función.

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

```
90    -- comprogar si cadena vacia
91    cadenaNoVacia :: [Char] -> Bool
92    cadenaNoVacia x = numwords > 0
93        where numwords = length (words x)
```

5.2.5. Definiciones locales

Formas de definir una función: definiciones locales

Es posible definir una función en cualquier punto de otra función.

Es muy importante que la definición esté algunos espacios a la derecha de la posición en la que empieza a definirse la función. En otro caso, Haskell mostrará un error.

```
95    -- comprobar si lista vacia
96    listaNoVacia :: [a] -> [a] -> Bool
97    listaNoVacia x y = numitems > 0
98        where numitems = length (x ++ y)
```

Realizar ejemplos en WINHUGS

RESTO DE LA CLASE

5.3.1 Practicar

Realizar ejemplos en WINHUGS

- Dada una lista de enteros, devuelve la lista de los sucesores de cada entero.

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

5.3.2 Practicar

Realizar ejemplos en WINHUGS

- Dada una lista de enteros, devuelve una lista con los elementos que son positivos

```
obtenerPositivos  :: [Integer] -> [Integer]
obtenerPositivos [] = []
obtenerPositivos (x:xs)
    | x>=0 = x : obtenerPositivos xs
    | otherwise = obtenerPositivos xs
```

5.3.3 Practicar

Realizar ejemplos en WINHUGS

- Redefinir la función until tal que until p f x aplica la f a x el menor número posible de veces, hasta alcanzar un valor que satisface el predicado p. until (>1000) (2*) 1 => 1024

```
hasta :: (a -> Bool) -> (a -> a) -> a -> a
hasta p f x =
    if p x then x
    else hasta p f (f x)
```

PRÁCTICA 2

5.4.1 Practica 2: Ejercicio 1

Ejercicio 1:

Definir la función comb tal que comb n k es el número de combinaciones de n elementos tomados de k en k; es decir:

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

```
Main> comb 6 2
15 :: Integer
```

5.4.1 Practica 2: Ejercicio 2

Ejercicio 2:

Definir la función raíces tal que raíces a b c es la lista de las raices de la ecuación $ax^2 + bc + c = 0$.

```
Main> raices_1 1 (-2) 1
[1.0,1.0] :: [Double]
Main> raices_1 1 3 2
[-1.0,-2.0] :: [Double]
```

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

5.4.1 Practica 2: Ejercicio 3

Ejercicio 3:

Los números de Fibonacci quedan definidos por las ecuaciones.

Definir la función que obtega la sucesión utilizando varias formas.

```
Main> fibonacci 5
8 :: Int
Main> fibonacci 40
165580141 :: Int
```

Los números de Fibonacci quedan definidos por las ecuaciones

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Esto produce los siguientes números:

- $f_2 = 1$
- $f_3 = 2$
- $f_4 = 3$
- $f_5 = 5$
- $f_6 = 8$
- $f_7 = 13$
- $f_8 = 21$

y así sucesivamente.

5.4.1 Practica 2: Ejercicio 4

Ejercicio 4:

Comprobar la pertenencia a una lista usando una función recursiva.

```
Main> pertenece 5 [5,4]
True :: Bool
Main> pertenece 1 [5,4]
False :: Bool
Main>
```