PRÁCTICA 4

INTRODUCCIÓN WINHUGS

Universidad de Huelva

4.1 REPASO Y EJEMPLOS P1

4.1. Repaso

WINHUGS

Operadores: (:) , (++) , (!!)

Funciones

head, tail, lenght, null, last, init, elem, notElem

lines, unlines, words, unwords, and, or , any, all

sum, producto, maximum, mínimum, repeat, replicate, cycle, iterate

map, filter, reverse, foldl, foldr, foldl1, foldr1, scanl, scanr, scanl1, scanr1, take, drop, takeWhile, dropWhile

4.1. Repaso

WINHUGS: ejemplos P1

esprimo: función que dado un entero positivo nos indica si dicho número es primo.

```
4   esprimo :: Int -> Bool
5
6   esprimo x = length (lista_divisores x x) == 2
7
8   lista_divisores :: Int -> Int -> [Int]
9
10  lista_divisores x y = filter(\z -> mod x z == 0) (take y (iterate (+1) 1))
11
12  -- Ejemplos para probar la funcion "esprimo"
13  -- esprimo 47 -> True
14  -- esprimo 24 -> False
```

Jesús Valeo Fernández

4.1. Repaso

WINHUGS: ejemplos P1

```
Main> take 10 (iterate (+1) 1)
[1,2,3,4,5,6,7,8,9,10] :: [Integer]
Main> filter (\z -> mod 11 z == 0)(take 11 (iterate (+1) 1))
[1,11] :: [Integer]
Main> lista_divisores 10 10 -- lista_divisores x x
[1,2,5,10] :: [Int]
Main> lista_divisores 11 11
[1,11] :: [Int]

Main> esprimo x = length (lista_divisores x x) == 2
```

4.1. Repaso

WINHUGS: ejemplos P1

unico: elimina los elementos repetidos

```
24
25  -- guarda en una lista todos los elementos diferentes de l
26
27  unico::(Eq a)=>[a]->[a]
28
29  unico [] = []
30
31  unico l = (head l):(unico (filter (\x -> x /= (head l)) l))
32
33  -- unico "aaaaaajallfqldfnlqhfclwqnfqwjefljwni" -> "ajlfqdnhcwei"
34
```

Ivan Bacho Delgado

4.1. Repaso

WINHUGS: ejemplos P1

unico: elimina los elementos repetidos

```
"a"++"j"++"l"++"fqdfnqhfcwqnfqwefwni"

"a"++"j"++"l"++"f"++"q"++"dnhcwnwewni"

…

"ajlfqdnhcwei"
```

4.1. Repaso

WINHUGS: ejemplos P1

Uso de foldl1: foldl1 (foldl1 (foldl1 (foldl1 (foldl1 (foldl1 (foldl1 1 3) 5) 7) 9) 11) 13) 15

```
12
13    -- EJEMPLO 4.
14    -- Repetir 5 veces el resultado de reducir una lista
15    -- aplicando la resta del 2º valor menos el 1º
16    -- cogiendo como valor inicial el 1º de la lista (con y sin replicate)
17    replicate 5 (foldl1(\x y -> (y - x) * 2) [1,3..15])
18    take 5 (repeat (foldl1(\x y -> (y - x) * 2) [1,3..15]))
19
```

Tomás Iglesias Sáez

4.1.2. Índice

¿Qué vamos a ver?

Tuplas

Funciones sobre tuplas

Creación de funciones

Ejercicios

4.2 TUPLAS

4.2.1. Tuplas

Una tupla es un tipo de datos formado por una secuencia ordenada de elementos con una estructura y tamaño fijo.

En Haskell las tuplas se representan entre paréntesis y separadas por coma. Por ejemplo, **(1, True).**

Una 2-tupla es una pareja de valores **(a, b)**. Una 3-tupla es un trío de valores **(a, b, c)**.

Son utilizadas cuando sabes exactamente cuantos valores tienen que ser combinados y su tipo depende de cuantos componentes tengan y del tipo de estos componentes.

Las tuplas pueden contener una combinación de valores de **distintos tipos**

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

4.2.2. Tuplas: funciones

Funciones sobre tuplas:

fst :: (a,b) -> a : Obtiene el primer valor de una 2-tupla.

```
Hugs> fst(1,2)+fst(2,1)
3 :: Integer
```

snd:: (a,b) -> a : Obtiene el segundo valor de una 2-tupla.

```
Hugs> fst(1,2)+snd(2,1)
2 :: Integer
```

4.2.2. Tuplas: funciones

Funciones sobre tuplas:

splitAt :: Int -> [a] -> ([a], [a]) : Divide una lista en un primer trozo de *n* elementos y un segundo trozo con el resto.

```
Hugs> splitAt 2 ['a','b','c','d']
("ab","cd") :: ([Char],[Char])
Hugs> splitAt 5 [1,2,3,4,5,6,7,8,9,10]
([1,2,3,4,5],[6,7,8,9,10]) :: ([Integer],[Integer])
Hugs> splitAt 2 ['a','b','c']
("ab","c") :: ([Char],[Char])
```

4.2.2. Tuplas: funciones

Funciones sobre tuplas:

span :: (a -> Bool) -> [a] -> ([a], [a]) : Divide una lista en un trozo con los primeros elementos que verifican una función y el resto de la lista.

```
Hugs> span (>"a") ["b","c","a"]
(["b","c"],["a"]) :: ([[Char]],[[Char]])
Hugs> span (<3) [1,2,4,5,6]
([1,2],[4,5,6]) :: ([Integer],[Integer])
Hugs> span (<"b") ["b","c","a"]
([],["b","c","a"]) :: ([[Char]],[[Char]])
```

4.2.2. Tuplas: funciones

Funciones sobre tuplas:

break :: (a -> Bool) -> [a] -> ([a], [a]) : Divide una lista en un trozo con los primeros elementos que no verifican una función y el resto de la lista.

```
Hugs> break (<"b") ["b","c","a"]
(["b","c"],["a"]) :: ([[Char]],[[Char]])
Hugs> break (<"b") ["a","b","c"]
([],["a","b","c"]) :: ([[Char]],[[Char]])
```

4.2.3. Tuplas: empaquetamiento

Funciones que empaquetan listas:

zip :: [a] -> [b] -> [(a, b)] : Toma dos listas y genera una lista emparejando los elementos de ambas. Si las listas no tienen la misma longitud, los elementos que sobran se pierden.

```
Hugs> zip [1,2,3] [9,8,7]
[(1,9),(2,8),(3,7)] :: [(Integer,Integer)]
Hugs> zip "abcde" [True,False]
[('a', True),('b', False)] :: [(Char,Bool)
Hugs> zip [] ["ab", "cd"]
[] :: [(a,[Char])]
```

4.2.3. Tuplas: empaquetamiento

Funciones que empaquetan listas:

zip3 :: [a] -> [b] -> [c] -> [(a, b, c)] : Es similar a *zip* pero tomando tres listas y generando tripletas.

```
Hugs> zip3 [3,5,7,9] [2,4,6] [1,0,6,8]
[(3,2,1),(5,4,0),(7,6,6)] :: [(Integer,Integer,Integer)]
Hugs> zip3 [3,5,7,9] [2,4] [1,0,6,8]
[(3,2,1),(5,4,0)] :: [(Integer,Integer,Integer)]
```

4.2.3. Tuplas: empaquetamiento

Funciones que empaquetan listas:

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c] : Es similar a *zip*, pero en vez de generar tuplas aplica la función sobre la pareja y almacena el resultado.

```
Hugs> zipWith (+) [3,5,4] [2,1,5,8]
[5,6,9] :: [Integer]
Hugs> zipWith (:) "eje" ["emp","los"]
["eemp","jlos"] :: [[Char]]
```

zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d] : Es similar a *zipWith* pero trabajando sobre tres listas

4.2.3. Tuplas: empaquetamiento

Funciones que empaquetan listas:

unzip :: [(a, b)] -> ([a], [b]) : Transforma una lista de parejas en una pareja de listas.

```
Hugs> unzip [(1,2),(2,3),(3,4)]
([1,2,3],[2,3,4]) :: ([Integer],[Integer])
```

unzip3 :: [(a, b, c)] -> ([a], [b], [c]) : Transforma una lista de tripletas en una tripleta de listas.

```
Hugs> unzip3 [(1,2,3),(2,3,4),(3,4,5)]
([1,2,3],[2,3,4],[3,4,5]) :: ([Integer],[Integer],[Integer])
```

4.2.5. Practicar

Realizar ejemplos en WINHUGS

10 MINUTOS

4.3 CREACIÓN DE FUNCIONES

4.3.1. Orden de prioridad: infix

Orden de prioridad

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

```
Hugs> 1 + 2
3 :: Integer
Hugs> + 1 2
ERROR - Syntax error in expression (unexpected token)
Hugs> (+) 1 2
3 :: Integer
Hugs> (<5) `map` [1,2,4,5,6]
[True,True,True,False,False] :: [Bool]
```

4.3.1. Orden de prioridad: infix

Orden de prioridad

En el caso de encadenar operadores no se define ninguna prioridad.

```
Hugs> 1 == 1 == 1
ERROR - Ambiguous use of operator "(==)" with "(==)“

Hugs> (1 == 1) == 1
ERROR - Cannot infer instance
*** Instance   : Num Bool
*** Expression : (1 == 1) == 1
```

4.3.1. Orden de prioridad: infix

Orden de prioridad

El resultado de evaluar la primera expresión es True. El error viene dado por la definición del operador "=="

```
Hugs> :info ==
infix 4 ==
(==) :: Eq a => a -> a -> Bool  -- class member
```

El operador se define para dos elementos del mismo tipo, que pertenecen a la clase **Eq** (equiparable).

4.3.2. Orden de prioridad: infixl

Orden de prioridad

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

```
Hugs> 1 - 2 - 1
-2 :: Integer
Hugs> (1 - 2) – 1  -- equivalente
-2 :: Integer
```

Si se quiere interpretar algo diferente es necesario utilizar los paréntesis

```
Hugs> 1 - (2 - 1)
0 :: Integer
```

4.3.2. Orden de prioridad: infixl

Orden de prioridad

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

```
x1 x2 ··· xn−1 xn => (. . . (x1 x2) ··· xn−1) xn
```

```
Hugs> foldl (\x y -> 2*x + y) 4 [1,2,3]
43 :: Integer
```

(2 * (2 * (2 * 4 + 1) +2) +3)

(2 * (2 * 9 + 2) +3)

(2 * 20 + 3)

43

4.3.3. Orden de prioridad: infixr

Orden de prioridad

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

```
Hugs> 2 ^ 1 ^ 2
2 :: Integer
Hugs> 2 ^ (1 ^ 2)
2 :: Integer
Hugs> (2 ^ 1) ^ 2
4 :: Integer
Hugs>
```

4.3.2. Orden de prioridad: infixl

Orden de prioridad

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

x1 x2 · · ·xn−1 xn => x1 (x2 · · ·(xn−1 xn) . . .)

foldl (\x y -> 2*x + y) 4 [1,2,3]

(2 * 1 + (2 * 2 + (2 * 3 + 4)))

(2 * 1 + (2 * 2 + (10)))

(2 * 1 + (14))

16

4.3.3. Orden de prioridad: “.”

Orden de prioridad

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

```
Hugs> succ . pred 4
ERROR - Cannot infer instance
*** Instance   : Enum (b -> a)
*** Expression : succ . pred 4
```

Error debido a que primero se realiza la operación pred 4 (de resultado 3). Después se intenta realizar la composición del número 3 con la función succ. Para poder realizar una composición de funciones son necesarias dos funciones. No es posible componer una función con un número. De esta forma, ponemos de manifiesto que entre un operador (sea cual sea) y una función, primero se evaluará la función.

4.3.3. Orden de prioridad: “.”

Orden de prioridad

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

```
Hugs> (succ . pred) 4
4 :: Integer
Hugs> (succ . succ) 'b'
'd' :: Char
Hugs> :info .
infixr 9 .
(.) :: (a -> b) -> (c -> a) -> c -> b
```

¿Cuál se evalúa primero?

4.3.3. Orden de prioridad

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`

4.3.3. Orden de prioridad: “.”

Evaluación perezosa

Los lenguajes que utilizan esta técnica sólo evalúan una expresión cuando se necesita

```
Main> soloPrimero 4 (7/0)
4 :: Integer
```

```
1  soloPrimero::a->b->a
2  soloPrimero x _ = x
```

El subrayado “_” denota que se espera un parámetro pero que no se necesita nombrarlo ya que no se va a utilizar en el cuerpo de la función.

La expresión 7/0 en Haskell tiene el valor Infinito. La evaluación de la expresión anterior provocaría un error en la mayoría de los lenguajes imperativos, sin embargo en Haskell no se produce un error debido a que el segundo argumento no llega a evaluarse por no ser necesario.

4.3.4. Funciones propias

NOTA: Notación (x:xs) -> cabecera:resto

```
2
3   suma :: [Int] -> Int
4   suma [] = 0
5   suma (x:xs) = x + sum xs
```

```
Hugs> :reload
Main> suma [1,2,3,4]
10 :: Int
Main> suma [1..20]
```

4.3.4. Funciones propias

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

4.3.4.1. 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)
```

4.3.4.2. Guardas “|”

Formas de definir una función: utilizando guardas

Factorial de un número entero:

```
15
16  {- FACTORIAL GUARDAS -}
17
18  --factorial_guarda ::Int->Int
19  --factorial_guarda :: (Num a, Ord a) => a -> a
20  factorial_guarda n
21      |n==0        =1
22      |n > 0       = n * factorial_guarda(n-1)
23      |otherwise   = error "valor negativo"
```

4.3.4.2. 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
```

4.3.4.3. 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
```

4.3.4.3. 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"
```

4.3.4.4. 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.

4.3.4.4. 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
```

4.3.4.4. CASE

Formas de definir una función: CASE

Mini calculadora

```
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
```

4.3.4.4. CASE

Formas de definir una función: CASE

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

4.3.4.5. 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
```

4.3.4.5. 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)
```

4.3.4.5. 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)
```