Programación II

Introducción a la programación funcional

Paradigmas de programación

Programación imperativa.

Los programas estás constituidos por sentencias que alteran el estado del mismo.

Las asignaciones producen una serie de efectos laterales que oscurecen la semántica del lenguaje (Referential opaqueness):

- es dificil demostrar la corrección de un programa
- no es posible evaluar expresiones en paralelo
- hay complicaciones para optimizar código

Programación funcional

El modelo funcional, tiene como objetivo la utilización de funciones matemáticas puras sin efectos laterales y, por tanto, sin asignaciones destructivas.

Características

- **Transparencia referencial:** el valor que devuelve una función está únicamente determinado por el valor de sus argumentos.
- **Funciones de orden superior:** permite que las funciones sean tratadas como valores de *primera clase*.
- **Inferencia de tipos y polimorfismo.**
- **Lazy evaluation:** una expresión no se evalua hasta que es necesario

Lenguaje Haskel

Conceptos básicos.

Modo interactivo: espera una expresión, luego la evalua y muestra el resultado

```
? (2+3)*8
40
? sum [1..10]
55
```

En los ejemplos anteriores, se utilizaron funciones estándar, incluidas junto a una larga colección de funciones en un fichero denominado *estándar prelude* que es cargado al arrancar el sistema.

Uso de funciones creadas por el usuario

El usuario puede crear un archivo con funciones para luego cargarlas y utilizarlas:

```
cuadrado::Integer -> Integer
cuadrado x = x * x

menor::(Integer, Integer) -> Integer
menor (x,y) = if x <= y then x else y
```

Luego hay que cargar el archivo mediante

```
:load fichero.hs
? cuadrado 9
81
```

Consideraciones sobre funciones

- En Haskel solo se aceptan funciones puras, para modelar funciones impuras se utilizan monadas.
- Las funciones son valores, pueden ser transferidas a otras funciones.
- Las funciones que reciben otras funciones como argumentos se denominan **funciones de orden superior**.
- Se admiten funciones anónimas...

```
listFunc = [\x y -> x + y,\x y -> x - y,\x y -> x * y,\x y -> x / y]

-- > (listFunc!!(1)) 4 5
```

Consideraciones sobre funciones

Las funciones pueden ser recursivas:

```
facR 0 = 1
facR n = n * facR (n - 1)
```

O no:

```
facNR n = product [1..n]
```

Definición de funciones

- En algebra una funcion se expresa como f : A -> B, donde A es el dominio y B el codomio o contradominio
- En haskell escribimos:

```
incr :: Int -> Int
incr n = n + 1
```

donde el Int a la izquierda de -> es el dominio y el Int a la derecha el rango

- La expresión a la derecha del símbolo **::** es el *tipo de la función*
- No es obligatorio indicar el tipo

Definición de funciones

- El operador **->** es asociativo a derecha

```
concatena :: String -> String -> String
concatena s1 s2 = s1 ++ s2
```

- La aplicación de funciones es asociativa a izquierda:

```
partialConcat :: String -> String
partialConcat = concatena "mundo"

-- > partialConcat "Hola"
```

La aplicación parcial de funciones se denomina **Currying**

Definición de funciones

- Una versión **uncurried** de concatena sería:

```
concatenau :: (String, String) -> String
concatenau (s1, s2) = s1 ++ s2
-- > concatenau ("hola ", "mundo")
```

- Los operadores infijos son en realidad funciones, entonces se pueden aplicar parcialmente:

```
concatenap s1 = (s1 ++)
-- > concatenap "Hola" "Mundo"
```

Composición de funciones

Al igual que en algebra, cuando el codominio de una función coincide con el dominio de otra, las mismas pueden ser compuestas

```
wordCount = length . words -- both function in Prelude
-- > wordCount "Hola Mundo, estas son seis palabras"
```

Pattern matching

Se trata de un mecanismo sintáctico que permite clarificar la definición de funciones cotejando sus argumentos con patrones.

```
primero (x,y) = x
segundo (x,y) = y
prim (x, _) = x
seg (_, y) = y
```

Se pueden aplicar a listas:

```
cab (x:xs) = x
cola (x:xs) = xs
-- > cab [1,2,3]
-- > cola [1..55]
```

Ejercicios

1. Definir una función recursiva que aplique las funciones almacenadas en listFunc a un par de argumentos.
2. Ingresar la siguiente lista en MiModulo.hs

```
mundos = [" lindo ", " feo ", " loco "]
```

Usando la función partialConcat, definir una función recursiva que genere como salida:

["mundo lindo", "mundo feo", "mundo loco"]

3. Genera otra versión de la función anterior, usando aplicación parcial sobre el operador de concatenación de listas (++).

Posibles soluciones:

```
aplica x y [] = []
aplica x y (h:c) = [h x y] ++ aplica x y c
aplica_funciones x y = aplica x y listFunc
```

```
mundos = [" lindo ", " feo ", " loco "]

f_mundos [] = []
f_mundos (x:xs) = [partialConcat x] ++ f_mundos xs
los_mundos = f_mundos mundos
```

```
f_mundos2 [] = []
f_mundos2 (x:xs) = [("mundo" ++) x] ++ f_mundos2 xs
los_mundos2 = f_mundos2 mundos
```

Guardias.

Se trata de un mecanismo sintáctico que facilita el entendimiento de funciones que usan múltiples condiciones

```
mm n m  | n > m = "El primero es mayor"
                | n < m = "El segundo es mayor"
            | otherwise = "Ambos son iguales"
```

Listas por comprensión.

Se trata de una notación sintáctica originada en la definición de conjuntos por comprensión que permite la generación de listas a partir de otras existentes.

```
cubo10 = [x^3 | x <- [1..10]]
```

Pueden usarse múltiples generadores separados por **,**

```
pares1 = [(x,y) | x <- [1..2], y <- [1..3] ]
pares2 = [(x,y) | y <- [1..3], x <- [1..2] ]
```

Generadores dependientes de alguno anterior

```
pares3 = [(x,y) | x <- [1..4], y <- [x..x+2] ]
```

Filtro en List comprehensions

```
factores n = [x | x <- [1..n], n `mod` x == 0 ]
```

Condiciones.

Se puede hacer uso de case para seleccionar entre posibles valores de un parámetro

```
suml l = case l of
                [] -> 0
                (h:t) -> h + suml t
```

Esto también se puede hacer utilizando **pattern matching**

```
suml2 [] = 0
suml2 (h:t) = h + suml t
```

Declaraciones locales.

Uso de let .. in

```
superfCirc radio = let pi = 3.14;
                                        r2 = radio * radio
                                   in pi * r2
```

Uso de where

```
superfCirc2 radio = pi * r2
                  where pi = 3.14; r2 = radio * radio
```

Ejercicios

4. Usando la notación de listas por comprensión, escribir una función que dada una condición y una lista, filtre los elementos que cumplan la condición. Por ejemplo:

filtro (>3) [5,1,7,0,9] = [5,7,9]

5. Dado un número y una lista ordenada, insertar el número en la posición correcta.

6. Explique el funcionamiento de la siguiente función:

funDesc ll = [x | l1 <- ll, x <- l1]

7. Escribir una función recursiva para multiplicar dos enteros usando adición

8. Escribe una función que acepte dos strings y cuente la cantidad de veces que aparecen los caracteres que componen el primer string, en el segundo. Por ejemplo:

cuentaC "az" "bvvvatsszaaaz" = 6

9. Escribe una versión alternativa de la función anterior

Posibles soluciones:

```
--4. Usando la notación de listas por comprensión, escribir una función
--que dada una condición y una lista, filtre los elementos
--que cumplan la condición. Por ejemplo:
-- **filtro (>3) [5,1,7,0,9] = [5,7,9]**
filtrar cond lis  = [x | x <- lis, cond x]
```

```
--5. Dado un número y una lista ordenada, insertar el número en la posición correcta.
inserta_ordenado n lis = [x | x <- lis, x<n] ++ [n] ++ [x | x <- lis, x>=n]
```

```
--6. Explique el funcionamiento de la siguiente función:
-- **funDesc ll = [x | l1 <- ll, x <- l1]**
funDesc ll = [x | l1 <- ll, x <- l1]
--Cada elemento de ll es una lista (l1). Genera una lista a partir de una
--lista de listas.
```

```haskell
--7. Escribir una función recursiva para multiplicar dos enteros usando adición
mult _ 0 = 0
mult 0 _ = 0
mult x y = x + mult x (y-1)
```

```haskell
--8. Escribe una función que acepte dos strings y cuente la cantidad
--de veces que aparecen los caracteres que componen
--el primer string, en el segundo. Por ejemplo:
-- **cuentaC "az" "bvvvatsszaaaz" = 6**
cuentaC ca1 ca2 = sum [1 | x <- ca1, y <- ca2, x==y]
```

```haskell
--9. Escribe una versión alternativa de la función anterior
cuentaC2  [] _ = 0
cuentaC2 (h:t) ca2 = length (filtrar (==h) ca2) + cuentaC2 t ca2
```

Sistema de tipos.

- Estáticamente tipado: errores detectados en tiempo de compilación.
- Los tipos pueden ser inferidos: Haskell no fuerza a indicar el tipo de una expresión.
- Las funciones son valores, por lo tanto, tienen un tipo asociado.

Tipos.

- Tipos básicos: Int, Bool, Char, etc.
- Tipos predefinidos: Maybe, [a]

Maybe a = Ñothing | Just a

[a] = [] | (:) a [a]

```
Main> (:) 3 ((:) 4 ((:) 5 []))
```

Tipos definidos por el usuario.

- data: Introduce nuevos tipos de datos

```
data Color = Rojo | Azul | Verde

esVerde Verde = True
esVerde _     = False
```

- type: renombra tipos existentes

```
type Nombre = String
```

Tipos recursivos.

- La definición de un tipo se llama a si misma

```
data ArbolBin a = Hoja a | Nodo (ArbolBin a) a (ArbolBin a)
        deriving (Eq,Show)

arbolEjemplo = Nodo (Nodo (Hoja 7) 1 (Hoja 6)) 5 (Nodo (Hoja 11) 15 (Hoja 4))
```

- Mediante **deriving** el sistema genera automáticamente instancias de los métodos definidos en las clases para el tipo que se está definiendo

Probar ejecutar sin la clausula deriving

Polimorfismo paramétrico.

- Los tipos de datos pueden estar parametrizados. En esta definición **a** es un parámetro de tipo.

```
data ArbolBin a = Hoja a | Nodo (ArbolBin a) a (ArbolBin a)
        deriving (Eq,Show)

arbolEjemplo = Nodo (Nodo (Hoja 7) 1 (Hoja 6)) 5 (Nodo (Hoja 11) 15 (Hoja 4))
```

Ejercicios

10. Define un tipo de datos de figuras geométricas (cuadrado, rectángulo, triángulo). Escribe una función que calcule la superficie de las mismas.

11. Define una función recursiva que imprima como una lista los elementos de un árbol de tipo ArbolBin a.

12. Define un árbol sin información en los nodos, solo en las hojas. Escribe una función que acepte como argumento un árbol de ese tipo y calcule la profundad máxima del mismo.

Posibles soluciones:

- Ejercicio 10

```
--10. Define un tipo de datos de figuras geométricas (cuadrado,
--rectángulo, triángulo). Escribe una función que calcule la
--superficie de las mismas.
data Figura =  Cuadrado   Float
                        |  Rectangulo Float Float
                        |  Triangulo  Float Float

superficie_figura ( Triangulo base altura) = base * altura / 2.0
superficie_figura ( Rectangulo base altura) = base * altura
superficie_figura ( Cuadrado lado) = lado * lado

-- > superficie_figura ( Triangulo 2 4 )
```

- Ejercicio 11

```
--11. Define una función recursiva que imprima como una lista los
--elementos de un árbol de tipo ArbolBin a.
lista_pre (Hoja h) = [h]
lista_pre (Nodo i v d) = [v] ++ lista_pre i ++ lista_pre d

-- Main > lista_pre arbolEjemplo
```

- Ejercicio 12

```
--12. Define un árbol sin información en los nodos, solo en las hojas.
--Escribe una función que acepte como argumento un árbol de
--ese tipo y calcule la profundad máxima del mismo.
data Arbol2 a = Hoja2 a | Nodo2 (Arbol2 a) (Arbol2 a)
        deriving (Eq, Show)

otro_arbol = Nodo2 (Nodo2 (Hoja2 7) (Hoja2 6)) (Nodo2 (Hoja2 11) (Hoja2 4))
otro_mas_corto = Nodo2 (Hoja2 11) (Hoja2 4)

prof (Hoja2 h) p = p + 1
prof (Nodo2 i d) p = max (prof i p + 1) (prof d p + 1)

profundidad a = prof a 0
```

Operadores de orden superior.

- foldr: encapsula el esquema de recursión visto en **suml2**

```
suml2 [] = 0
suml2 (h:t) = h + suml t
```

El uso es el siguiente:

```
sumatoria = foldr (+) 0
productoria = foldr (*) 1
```

Operadores de orden superior.

- map: Aplica una función pasada como argumento a los elementos de una lista

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (h:t) = f h : map f t
Main > map (\x -> x*x) [1,2,3]
```

Ejercicios

13. La función concat en el Prelude concatena en una lista, las listas que constituyen una lista dada como argumento (lista de listas). Escribe una versión propia de concat usando foldr.

14. Prueba el funcionamiento de la función takeWhile del Prelude. Escribe una versión propia de esta función aplicando recursión explícita.

15. Cree una verión de filtrar que utilice map en lugar de listas por comprension

Posibles soluciones:

- Ejercicio 13

```
--13.  La función concat en el Prelude concatena en una lista, las
--listas que constituyen una lista dada como argumento (lista de
--listas). Escribe una versión propia de concat usando foldr.

concat2 = foldr (++) []
```

- Ejercicio 14

```
--14. Prueba el funcionamiento de la función takeWhile del Prelude.
--Escribe una versión propia de esta función aplicando recursión explícita

-- Main > takeWhile (\x -> x >= 3) [3,4,1,5,6]
-- [3,4]

takeWhileR f [] =  []
takeWhileR f (h:t) = if f h then [h] ++ (takeWhileR f t) else []
```

- Ejercicio 15

```
--15. Cree una verión de filtrar que utilice map en lugar de listas por comprension

filtrar2 cond lista = concat(map (\x -> if cond x then [x] else []) lista)
```

Bibliografía y enlaces útiles.

- Labra, Jose: Introducción al lenguaje Haskell: http://www.x.edu.uy/inet/IntHaskell98.pdf
- Hughes, John: Why Functional Programming Matters: http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf
- Pagina oficial de Haskell: http://www.haskell.org/haskellwiki/Haskell