PARADIGMAS DE PROGRAMACIÓN
Licenciatura en Informática
Programador Universitario

UNIDAD I
PARADIGMAS DE PROGRAMACIÓN
HASKELL
Paradigma Funcional

Paradigma Funcional

Lenguajes Funcionales son tipo → **Intérprete**

constan de

- **Objetos de datos**, de tipo atómico, listas, ...
- **Funciones Primitivas**: de control, de estructuras, aritméticas....
- **Formas funcionales** para crear y combinar funciones
- **Mecanismos** para aplicar los argumentos y producir valores de funciones

2

HASKELL

a) Objetos de datos

. **Átomos**: cadenas de números o caracteres

```
28, “programacion”, -27, 8.3
```

. **Identificadores**: átomos no numéricos. Se usan como nombres de funciones o variables.

```
x, doble,  y25
```

. **Listas**: átomos y/o sublistas encerrados entre corchetes.

```
[20, 4, 35] [[x, y], [a, b, c, d, e], [f]]
```

Las listas tienen que ser del mismo tipo

3

HASKELL

b) Funciones primitivas predefinidas:

i) Tratamiento de listas

head - Devuelve la cabeza de la lista

```
head [1,2,3,4] --> 1
```

tail - Devuelve la cola de la lista

```
tail [1,2,3,4] --> [2,3,4]
```

last - Devuelve el último elemento de la lista

```
last [1,2,3,4] --> 4
```

init - Devuelve la lista sin el último elemento

```
init [1,2,3,4] --> [1,2,3]
```

4

HASKELL

i) Tratamiento de listas

take - Extrae n elementos del comienzo de la lista

```
take 2 [1,2,3,4] --> [1,2]
```

drop - Elimina los primeros n elementos de la lista

```
drop 2 [1,2,3,4] --> [3,4]
```

++ - Concatena dos listas

```
[1,2,3,4] ++ [5,6,7] --> [1,2,3,4,5,6,7]
“Hola” ++ “ ” ++ “mundo” --> “Hola mundo”
```

!! - Permite obtener el elemento de la lista que se encuentra en la posición del índice. Empieza desde 0

```
[1,2,3,4] !! 2 --> 3
```

5

HASKELL

i) Tratamiento de listas

length - Devuelve la cantidad de elementos de la lista
null - Test para determinar si la lista está vacía
maximum - Devuelve el elemento mas grande de la lista
minimum - Devuelve el elemento mas pequeño de la lista
sum - Devuelve la suma de los elementos de la lista
product - Devuelve el producto de los elementos de la lista
elem - Determina si un elemento pertenece a la lista

6

HASKELL

ii) Asignación, relaciones booleanas y predicados

let – Asigna un valor a un identificador.

```
let z = [10,20,30]
```

== , /= , not , && , ||

```
1+2 == 4 --> False
1+2 /= 4 --> True
not(5==0) --> True
3==3 && “hola”== “hi” --> False
3==3 || “hola”== “hi” --> True
```

[a..b] - Genera una lista que contiene números desde a hasta b

```
[1..7] --> [1,2,3,4,5,6,7]
[0,3..10] --> [0,3,6,9]
[1..] --> [1,2,3,4,…] lista ∞
```

iii) Artiméticas

```
+ - * / mod div
```

7

HASKELL

iv) Condicionales

La sentencia if es una expresión que se evalúa y retorna un valor, por lo que la rama else es obligatoria

```
let a = (if x > 100  then x  else x*2) + 2
```

```
describeLista :: [a] -> String
describeLista [] = “Lista vacia”
describeLista [x] = “Lista con un elemento”
describeLista [x|_] = “Lista con mas elementos”
```

```
describeLista’:: [a] -> String
describeLista’ xs = “Lista” ++ case xs of
                                  [] -> “ vacia”
                                  [x] -> “ con un elemento”
                                  xs  -> “ con mas elementos”
```

8

HASKELL

c) Formas funcionales para crear y combinar funciones

- El nombre de una función debe comenzar con **minúscula**
- Una función sin parámetros es una definición

```
saludo = “Hola mundo”
```

- Una función con parámetros puede definir el tipo de cada uno
- El sistema de tipado de Haskell (strong static typing) infiere los tipos

```
saludarA x = “Hola ” ++ x
> :t saludarA
saludarA :: [Char] -> [Char]
```

aún así, podemos definir una función con parámetros conjuntamente con su tipo

HASKELL

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = product [1..n]
```

```
length'::[a] -> Int
length' [] = 0
length' (_:xs) = 1 + length' xs
```

```
sumaLista ::[Int] -> Int
sumaLista [] = 0
sumaLista (x:xs) = x + sumaLista xs
```

Pattern Matching

Pattern Matching consiste en especificar los patrones a los que deben ajustarse algunos datos y luego verificar para ver si lo hace y deconstruir los datos de acuerdo con esos patrones

```
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Cuando llame a fibonacci, los patrones se verificarán de arriba a abajo y cuando se ajuste a un patrón, se usará el cuerpo de la función correspondiente. La única forma en que un número puede ajustarse al primer patrón aquí es si es 0. Si no lo es, cae en el segundo patrón, y así sucesivamente hasta que alguno coincida, sino da error.

```
Non-exhaustive patterns in function XXXX
```

11

Pattern Matching

- El patrón (x:xs) se usa mucho en Haskell, especialmente con funciones recursivas.

```
pertenece [] y = False
pertenece (x:xs) y = if y==x then True else pertenece xs y
```

OBS: Los patrones que tienen **:** en ellos solo coinciden con listas de longitud 1 o más.

12

Guards

Mientras que los patrones son una forma de asegurarse de que un valor se ajuste a alguna forma y deconstruirlo, las *guards* son una forma de comprobar si alguna propiedad de un valor (o varias de ellas) es V o F

```
fibonacci’ n
 | n <= 0 = 0
 | n == 1 = 1
 | otherwise = fibonacci' (n-1) + fibonacci' (n-2)
```

```
fibonacciE n =
  let fibAux n acc1 acc2
   | n <= 0 = acc1
   | otherwise = fibAux (n-1) acc2 (acc1+acc2)
  in fibAux n 0 1
```

```
fibonacciE' n = fibAux' n 0 1
    where fibAux' n acc1 acc2
       | n <= 0 = acc1
       | otherwise = fibAux' (n-1) acc2 (acc1+acc2)
```

13

List Comprehension

Derivan del concepto matemático *set comprehension* que permite construir a partir de conjuntos generales conjunto más específicos.

Ejemplo:
Obtener los primeros 10 números pares $C = \{ 2x \mid x \in [1,10]\}$

```
take 10 [2,4..]
```

```
diezPrimerosPares = [2*x | x <- [1..10]]
> diezPrimerosPares --> [2,4,6,8,10,12,14,16,18,20]
```

```
paresMenoresQue z = [2*x | x <- [1..z], 2*x < z]
> paresMenoresQue 8 -->[2,4,6]
```

```
eliminarMinusculas :: [Char] -> [Char]
eliminarMinusculas st = [c | c <-st, c `elem` ['A'..'Z']]
> eliminarMinusculas “SuperCalifragilisticoEspialidoso” --> “SCE”
```

14

High Order Programming

Las funciones de Haskell pueden tomar funciones como parámetros y devolver funciones como valores de retorno. Una función que hace cualquiera de esos se llama **Función de Orden Superior**.

- Genericidad

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

- Lambdas

```
map (\(a,b) -> a + b) [(1,2),(3,4),(5,6)]
```

- Composición

```
g x = x^2
f x = x + 1
fg = f . g
```

15

Imperativo vs Funcional

```
FUNCION maximo(vector)
  SI (tamaño(vector) = 0) ENT.
       error “vector vacío”
  max ← vector[0]
  i ← 1
  MIENTRAS (i < tamaño(vector))
       SI (max < vector[i]) ENT.
          max←vector[i]
       i ← i + 1
  RETORNA max
```

```
maximo []= error “vector vacio”
maximo [x]= x
maximo (x:xs)
 | x > maxTail = x
 | otherwise = maxTail
where maxTail  = maximo xs
```

```
maximo’ []= error “vector vacio”
maximo’ [x]= x
maximo’ (x:xs)= max x (maximo’ xs)
```

```
FUNCION sumaMult(vector)
  n ← tamaño(vector)
  SI ( n = 0) ENT.
       error “vector vacío”
  acc ← 0
  HACER n VECES (para i=1,..n)
       acc← acc + 3*vector[i]
  RETORNA acc
```

```
sum (map (3*) xs)
```

16

Lenguaje Haskell

- **Representativo del Paradigma Funcional**
- **Evaluación tardía**
- **Estáticamente Tipado**
- **Inferencia de tipo**
- **Elegante y conciso**

17