Haskell

Haskell, Paul Hudak 1987 en honor a Haskell Curry

Lenguaje funcional **puro, no estricto y fuertemente tipado**.

* **Puro**: no efectos laterales, puras expresiones.
* **No estricto**: usa un orden no aplicativo (normal, evaluación perezosa)
* Haskell no evalúa nada si no necesito desplegar el resultado de la expresión.
* **Tipaje fuerte**: todas las expresiones tienen un tipo de dato.

Programas en Haskell

Consiste en declaraciones y definiciones de funciones.

* Declarla: se indica el tipo



* Definirla: dar el método de cómputo.



Ejemplo



sucesor : :  Integer -> Integer

sucesor x = x + 1



Programacion Haskell

* El nombre de las funciones debe empezar en minúscula.

**Currificacion**

Todas las funciones de más de un argumento están currificadas.

**Debemos usar let para usar las funciones currificadas.**

let prueba = sumaCuadrados 4 = 16

prueba 2

20

Ejemplo

Calcula la suma de los cuadrados de dos argumentos.

sumaCuadrados : : Int -> Int -> Int



sumaCuadrados x y = x \* x + y \* y

¿Qué regresa sumaCuadrados 2? Una función que calcula el cuadrado de y y se le sumaría 4.



Recursion y patrones en Haskell

Ejemplo: Calculo de factorial

factorial : : Integer -> Integer

factorial 0 = 1

factorial n = n \* factorial(n – 1), sin los paréntesis, se cicla.

Listas

Se representan como [2, 3, 4], las listas son homogéneas, del mismo tipo.

Head

Head : : [a] -> a

Head : : (x:lista) = x

Tail

Tail : : [a] -> a

Tail : : (\_:xs) = xs

Polimorfismo



Ejemplo



length : : [a] -> Int



length [] = 0

length (x:xs) = 1 + length xs

String

En Haskell los strings son listas de carecteres = [‘h’, ‘i’]

* Función reverse “coche” 🡺 “ehcoc”

Funciones de listas comunes

* **LET** 
  + let a = 1, es lo mismo que escribir a = 1 en un script y después cargarlo.
* **APPEND**
  + Operador binario
  + [1, 2] ++ [3, 4] = [1, 2, 3, 4]
* **CONS**
  + ‘A’**:** “ SMALL CAT”
  + “A SMALL CAT”
* **GET Nth**
  + [1, 2, 3, 4, 5] !! 1
  + 2 (Empiezan en 0th)
* **INIT**
  + Elimina el ultimo elemento de la lista
  + init [1, 2, 3]
  + [1, 2]
* **NULL**
  + null []
  + True, constante predefinida
* **MINIMUM Y MAXIMUM**
  + minimum [8, 4, 10]
  + 4
  + maximum [1, 9, 10]
  + 10
* **SUM Y PRODUCT**
  + Sum [8, 4, 10]
  + 22
  + Product [8, 4, 10]
  + 320
* **elem**
  + 4 `elem` [3, 4, 5]
  + Elem 4 [3,4,5]
  + True

Condiciones

Ejemplo: Posicion de un entero en una lista

Posint : : Int -> [Int] -> Int

Posint \_ [] = -1

Posint x (y:lista) =

If x == y then 1



Else 1 + posint x lista



Manejo de errores



Posint : : Int -> [Int] -> Int

Posint \_ [] = **error** “Enterno no encontrado”

Posint x (y:lista) =

If x == y then 1

Else 1 + posint x lista

Ejemplo: Obtener el primer elemento de una lista

car : : [a] -> a

car xs = **case** xs of

[] -> error “lista vacia”

(x:\_) -> x

Tipos de datos

* Atomicos y funcionales
  + Int, Integer, Char, Float, Double
* Estructurados
  + Listas [Int], …
  + Tuplas (‘b’, 4) ..
  + Combinaciones (“Maria”, [77, 90, 98])

Diagram

Description automatically generated

Represantaciones especiales

[1 . . 4] => [1, 2, 3, 4]

[a . . d] => [a b c d]

Salto

[1, 4 . . 12] => [1, 4, 7, 10]

Salto = 4 – 1 = 3

Firmas para funciones con variables de tipo (Variable polifórmica)

A picture containing diagram

Description automatically generated

Tuplas

[[Int]] == lista de listas de enteros

**([**Char], (Char, Char)**)** == tupla

(“hola”, (‘a’,’b’))

Ejemplo

Text, letter

Description automatically generated

Declaración de tipos de datos

Tipos de datos definidos por el usuario

**data** Color = Rojo | Verde | Azul 🡪 Rojo OR Verde OR Azul

Rojo : : Color

* Son funciones
* No se pueden repetir
* Tambien pueden tener argumentos

**data** punto a = pt a a

Pt 2.0 3.0 : : Punto Double

Se debe agregar **deriving (Show)** para crear una instancia de la clase Show y poder desplegarlo

Tipos recursivos

Arboles binarios

* e actua como una variable de tipo
* Se usa el constructor nodo

Data Arbol e =

Nodo (Arbol e) e (Arbol e)

| ArbolVacio

Listas Anidadas

Data ListaAnidada a =

Elemento a

| Lista [ListaAnidada a]

Ejemplos

Entrege el subárbol izquierdo del árbol

Izq : : Arbol a -> Arbol a

Izq (Nodo l n r) = l

Contar cuantos nodos tiene un árbol

numNodos : : Arbol a -> Int

numNodos ArbolVacio = 0

numNodos (Nodo l n r) =

1 + (numNodos l) + (numNodos r)

Recorrido

Recorrido : : Arbol a -> [a]

Recorrido ArbolVacio = []

Recorrido (Nodo l n r) =

(recorrido l) ++ [n] ++ (recorrido r)

Funciones currificadas

Cada función en Haskell solo toma un parámetro, para utilizar más de un parámetro se usan funciones currificadas.



Ejemplo



multThree : : (Num a) => a -> a -> a -> a



multThree x y z = x \* y \* z



Cualquier función currificada es de orden superior

Secciones

Las funciones infijas también se pueden aplicar parcialmente mediante secciones

Se rodea con paréntesis la función y solo se proporciona un parámetro en uno de los lados.

Ejemplos

esMayuscula : : Char -> Bool

esMayuscula = (`elem` [‘A’ … ‘Z’])

Funciones Lambda

Funciona igual que en Racket

Ejemplo

\ x -> x \* x



Se aplican igual

(\ x -> x \* x) 5 = 25

Ejercicio

Hacer una expresión lambda que multiplique los componentes de un par

(\x -> **fst** x \* **snd** x) (2, 3) => 6

Ejemplo **Flip**

La función voltea invierte el orden de los argumentos de una función binaria



Voltea : : (a -> b -> c) -> b -> a -> c



Voltea = \f x y -> f y x

Funciones de orden superior PREDEFINIDAS

Reciben funciones como argumentos o regresan funciones como resultados

* **Map**
  + Aplica una función a todos los elementos de una lista y regresa el resultado en otra lista.
  + Recibe una función f: a -> b
  + Recibe otro argumento que es una lista de a
  + Regresa como resultado una lista b
  + Map : : (a -> b) -> [a] -> [b]
  + map sqrt [1, 2, 3, 4] = [1, 1.41, … ]



¿Que pasa si tenemos dos listas?

Zip = regresa una lista de pares correspondientes, el primero con el primero, …



Ejemplo con map

Obtener el producto punto de dos arreglos dados como listas

Producto [1,2,3] [4, 5, 6] => 32

Producto : : [Int] -> [Int] -> [Int]

Producto l1 l2 =

Sum (map (\(p,s)->p\*s) (zip l1 l2))

* **All, any**
  + Aplica un predicado a todos los elementos de una lista y determina si todos lo cumplen.
  + All :: (a->Bool) -> [a] -> Bool

Ejemplo

All even [1, 2, 3] => False (AND)

Any even [1,2,3] => True (OR)

* **Filter**
  + Genera una lista con los elementos que cumplan el predicado que se recibe como parámetro.
  + Filter even [1, 2, 3, 4] = [2,4]
* **Foldl, foldl1 (Por la izquierda**)
  + Foldl = Reduce una lista aplicando un operador binario de izquierda a derecha, utilizando como primer operando su segundo argumento.
  + Foldl1 = Lo mismo, pero sin valor inicial.

Ejemplo



foldl (-) 1 [4,2,3] => 1-4-2-3 = -8



Foldl1 (/) [1,2,3] => 1/2/3

* **Foldr, foldr1**
  + De derecha a izquierda

Ejemplo

Foldr (-) 1 [4,2,3] => (4-(2-(3-1))) = 4



Foldr1 (/) [8,4,2] => (8/ (4/2)) = 4.0



* **Compose (.)**
  + Toma dos funciones y las encadena
  + (f.g.h) x 🡺 f(g(h(x))
  + Ejemplo
  + (sqrt.sum) [1,2,3,4,5] => 3.87298



**Ejemplo**

Calcula la suma de los valores absolutos de los elementos de una lista

Sumabs [-1, -2, -3] = 6

Sumabs = sum . map abs

* **Until**
  + Toma un predicado unario, una función unaria y un valor, reaploca la función al valor hasta que cumple el predicado.
  + Until (> 5) (+ 2) 0 = 6

Evaluación perezosa

Guardias

Se usan en definiciones por casos

Ejemplo:

Fact1 : : Int -> Int

Fact1 n

| n == 0 = 1



| n > 0 = n \* fact1 (n-1)



Ejemplo extraer pares

Pares : : [Int] -> [Int]

Pares [] = []

Pares (x:xs) =

| even x = x : pares xs



| otherwise = pares xs



Comprension de listas, combinacion de filter y map

**[f x | x <- xs]**

* X <-xs es el generador porque es el que esta sacando los valores de la lista, a los que se les puede aplicar una función f.

Ejemplo

prodcart [1,2] [3,4]

prodcart : : [a] -> [b] -> [(a,b)]

prodcart xs ys = [ (x, y) | x <- xs, y <- ys]

Ejemplo

Filtrar una lista de números positivos

Positivos [-3..3] => [1,2,3]

Positivos : : (Ord a) => [a] -> [a]



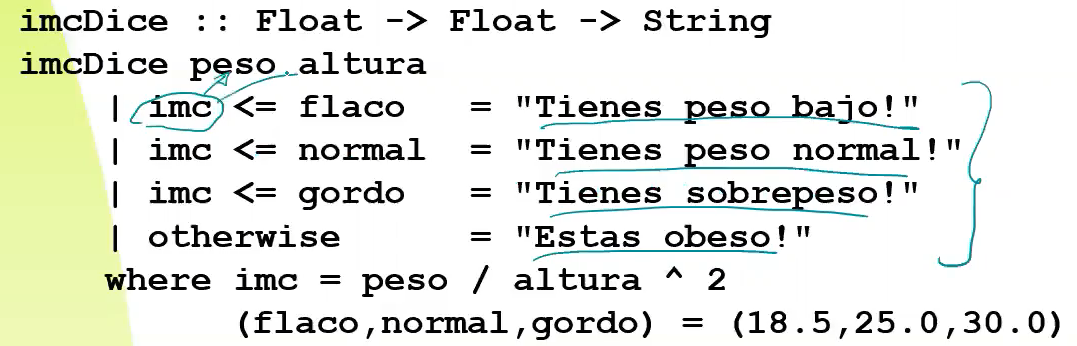
Positivos l = [x | x <- 1, x > 0]



WHERE Y LET

**WHERE**

Construcción sintática que permite definir localmente constantes y funciones y asociar nombres mediante empatamiento de patrones.





También le puedes pasar una función constante a where

calcIms : : [(Float, Float)] -> [Float]

calIms personas = [imp p a | (p, a) <- personas] where imc p a = p / a ^ 2

Ejemplo Quick Sort

Text

Description automatically generated



**LET**

Expresión que permite definir localmente constantes y funciones y asociar nombres mediante empatamiento de patrones.

***Let <bindings> in <expresión>***

Cilindro : : Float -> Float -> Float

Cilindro radio altura =

Let areaLado = 2 \* pi \* radio \* altura

areaTope = pi \* radio ^ 2

in areaLado + 2 \* areaTope

calcIms : : [(Float, Float)] -> [Float]

calcIms personas =

[imc | (peso, altura) <- personas, let imc = peso / altura ^2, imc >=25.0]

Estructuras infinitas

[1..] es una estructura infinita

[1,3..] también es un conjunto infinito de impares positivos

**Take** 6 [1..] = [1,2,3,4,5,6], funciona porque solo evalúa los primeros 6 términos de la lista por la evaluación perezosa.

Lista infinita de unos

Unos : : [Int]

Unos = 1:unos

Factorial con listas infinitas

Factorial :: Int -> Int

Factorial n = product (take n [1..]) 🡺 1\*2\*3…n

Fibonnaci

Fibo = 1:1:[a+b | (a,b) <- zip fibo (tail fibo)]