Tutorial Haskell

David Julián Guzmán Cárdenas
Cristian Alexanther Rojas Cárdenas
Luis Ernesto Gil Castellanos

UNIVERSIDAD NACIONAL DE COLOMBIA

Agenda

1. Introducción
2. Aspectos básicos del lenguaje
3. Tipos y clases de tipos
4. Sintaxis en Funciones
5. Recursión
6. Funciones de orden superior

Introducción

Motivación

- ¿Qué es Haskell?
  - Lenguaje de programación completamente funcional, moderno (1990, último estándar del 2010). ¿Es difícil?
- Imperativo vs Funcional
  - Secuencia de comandos vs cálculo de una expresión
  - Hojas de cálculo, valor en la celdas como expresiones
  - Énfasis en ¿QUÉ? se calculará y no en el ¿CÓMO?
  - Funcional: fácil de entender, preciso, corto.
  - *lazyness*.
- ¿Por qué Haskell?
  - Es adecuado para programas que necesiten ser altamente modificables y mantenibles.
  - Fuertemente tipado(no se puede tratar un entero como un puntero, ni seguir un puntero nulo)
  - Todos los compiladores son gratuitos (GHC, UHC, Hugs)
- ¿Quién usa Haskell?
  - LOLITA(Procesador de lenguaje natural por la universidad de Durham, 50k Haskell)
  - GHC

Aspectos básicos del lenguaje

Un nuevo comienzo

```
Prelude> 3 + 2.5
5.5
Prelude> 14 / 3
4.666666666666667
Prelude> not True || True
True
Prelude> 4 - 3 * 2
-2
Prelude> ( 4 - 3 ) * 2
2
Prelude> 5 /= 5
False
Prelude> "hello" == "hola"
False
Prelude> "Hola" >= "hola"
False
Prelude> "HOLA" >= "hola"
False
Prelude> "hola" >= "Hola"
True
Prelude>
```

Aritmética simple, precedencia

Lógica, operadores relacionales
{==, /=, >, <, <=, >=}

Aspectos básicos del lenguaje

Un nuevo comienzo

Funciones definidas por el usuario.

Se puede crear un script y cargarlo

No importa el orden de las funciones

```
Prelude> 14 / 7
2.0
Prelude> 14 `div` 7
2
Prelude> div 14 7
2
Prelude> div 14 3
4
Prelude> 14 / 3
4.666666666666667
Prelude> succ 9
10
Prelude> pred 'Z'
'Y'
Prelude> max 2.3 2.2229
2.3
Prelude> min (-2) (-5)
-5
Prelude>
```

Funciones infijas y prefijas

Máxima precedencia

Aspectos básicos del lenguaje

Un nuevo comienzo

```
if <expresión_relacional> then
        ...
        <acciónes_si>
        ...
else
       ...
       <acciones_sino>
       ...
```

Todo es una expresión, una expresión es un pedazo de código que devuelve algo, la sentencia if es una expresión dado el else obligatorio.

Los nombres de funciones pueden tener apóstrofe (‘), no pueden comenzar por mayúsculas.

Si una función no tiene parámetros, se llama definición.

```
let <nombre> <x1 x2 ... xn> =
<expresión>
```

Aspectos básicos del lenguaje

Listas

Es la estructura de datos más usada, son homogéneas.

Operaciones entre listas:

- concatenación (al final y al inicio)
- extraer elemento de una posición
- longitud
- ¿está vacía?
- tomar un trozo
- sumar, multiplicar
- máximo, mínimo
- ¿un elemento está en la lista?


```
[]
[[]]
[[], [], []]
```

Aspectos básicos del lenguaje

Rangos

Secuencias aritméticas de elementos que pueden ser enumerados (como los números y los caracteres), es una forma de hacer listas finitas e infinitas.

```
Prelude> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [1,2..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
Prelude> [1,3..20]
[1,3,5,7,9,11,13,15,17,19]
Prelude> take 10 [10,20..]
[10,20,30,40,50,60,70,80,90,100]
Prelude> take 5 (repeat 3)
[3,3,3,3,3]
Prelude> take 5 (cycle [1,2,3])
[1,2,3,1,2]
Prelude>
```

Aspectos básicos del lenguaje

Comprensión de listas

$$S = \{ 2 \cdot x \mid x \in \mathbb{N},\ x \leq 10 \}$$

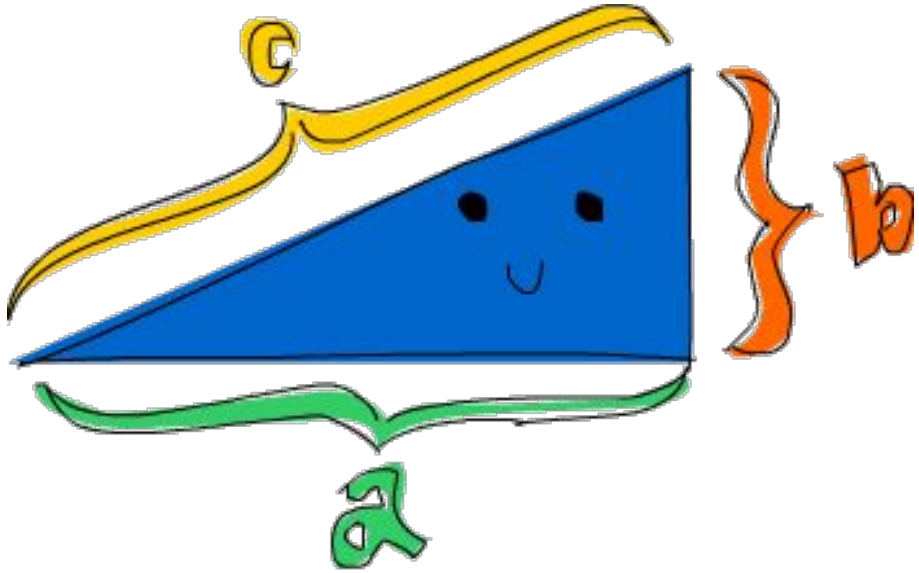
```
Prelude> [2*x | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude>
```

Ahora, números entre 50 y 100 que sean congruentes con 3 módulo 7

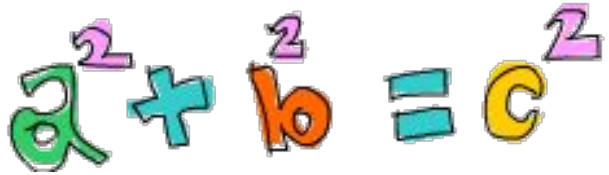
Aspectos básicos del lenguaje

Tuplas

- No tienen que ser homogéneas
- No hay tuplas singleton
- Se pueden comparar, siempre que sus tamaños sean iguales
- función **`zip`**
- **`zip`** toma el menor tamaño si son diferentes



¿Qué triángulo rectángulo que tiene aristas de dimensión entera, que todas sean menores a 10 y su perímetro sea 24?



Tipos y clases de tipos

Características de los tipos de datos en Haskell

- Estático:

  El tipo de cada variable es conocido al momento de compilar, lo cual implica código más seguro.

- Inferencial:

  No es necesario especificar el tipo de dato de una variable, Haskell puede inferir por su cuenta de nuestras funciones y expresiones.

Tipos y clases de tipos

Tipos en expresiones

- :t - > Comando que retorna el tipo de una variable.
- :: -> Implica que el dato es del tipo dicho, “es de tipo”.

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t True
True :: Bool
Prelude> :t "HELLO!"
"HELLO!" :: [Char]
Prelude> :t (True, 'a')
(True, 'a') :: (Bool, Char)
```

Tipos y clases de tipos

Tipos en funciones

- Para una función que tiene como entrada un parámetro de tipo Char y retorno otro parámetro de tipo Char la implementación es la siguiente.

```
1   removeNonUppercase :: [Char] -> [Char]
2   removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

- En el caso de una función la cual tiene mas de una parámetro de entrada, para inicializarla con tipos explícitos se tiene.

```
1   addThree :: Int -> Int -> Int -> Int
2   addThree x y z = x + y + z
```

Tipos y clases de tipos

Tipos de variables

- Int:

  Números de tipo entero limitados a 32 bits.
- Integer:

  Números de tipo entero sin limite de representación.
- Float:

  Números reales de punto flotante de precisión simple.
- Double:

  Números reales de punto flotante de precisión doble.
- Bool:

  Solo puede tener valores de tipo True o False.
- Char:

  Representación de un carácter.

Tipos y clases de tipos

Tipo Genérico

Cuando una función no se le es especificado el tipo de entrada y salida Haskell las toma como genéricas, es decir que puede ser cualquier tipo de dato de los mencionados anteriormente.

Gran parte de las funciones internas de Haskell están implementadas de esta forma.

```
Prelude> :t head
head :: [a] -> a
Prelude> :t tail
tail :: [a] -> [a]
Prelude> :t fst
fst :: (a, b) -> a
```

Tipos y clases de tipos

Clases de tipo

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Restricción de Clase

Una clase de tipo puede considerarse una forma de interfaz que define algún comportamiento. Las restricciones de clase tienen el fin de acotar los tipos tomado de forma genérica y así aplicar la función a valores con los cuales tenga sentido.

En el caso del ejemplo se clase de tipo ‘Eq’ provee una interfaz realizar comparaciones de igualdad, cualquier tipo que tenga sentido realizar una comparación de igualdad entre dos valores de ese mismo tipo deberían ser miembros de la clase Eq.

Tipos y clases de tipos

Algunas clases de tipo

ENUM

```
Prelude> :t (*)
(*) :: Num a => a -> a -> a
```

ORD

```
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
```

Sintaxis en funciones

Emparejamiento de patrones

Una función puede ser definida a partir de cuerpos de diferentes funciones para distintas entradas.

El funcionamiento del emparejamiento por patrones, al momento de recibir una entrada la compara con cada uno de los patrones, en el momento que coincida con algun patron, ejecuta los comandos correspondientes y termina la función.

```
1  sayMe :: (Integral a) => a -> String
2  sayMe 1 = "One!"
3  sayMe 2 = "Two!"
4  sayMe 3 = "Three!"
5  sayMe 4 = "Four!"
6  sayMe 5 = "Five!"
7  sayMe x = "Not between 1 and 5"
```

Sintaxis en funciones

Emparejamiento de patrones

Un emparejamiento de patrones no solo se limita a bifurcar el proceso de acuerdo al tipo de entrada, puede ser usado para implementar funciones recurrentes.

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

Además la condición del patrón puede ser también obtenido de variables de tipo tupla o lista.

```haskell
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

Sintaxis en funciones

Emparejamiento de patrones

Un caso de aplicación de emparejamiento de patrones sobre listas se observa cuando es necesario cambiar el comportamiento de la función de acuerdo a la cantidad de valores de la lista.

```
1  tell :: (Show a) => [a] -> String
2  tell [] = "The list is empty"
3  tell (x:[]) = "One element: " ++ show x
4  tell (x:y:[]) = "Two elements: " ++ show x ++ " " ++ show y
5  tell (x:y:_) = "Too long."
```

En el caso de la función sumatoria se tiene un ejemplo de recursión sobre listas.

```
1  sum' :: (Num a) => [a] -> a
2  sum' [] = 0
3  sum' (x:xs) = x + sum' xs
```

Sintaxis en funciones

Guards

Los guardias son funciones que tambien bifurcan de acuerdo al valor de entrada, no obstante, en este caso se permite aplicar expresiones sobre los valores de entrada y bifurcar de acuerdo a comparaciones lógicas.

```
1  bmiTell :: (RealFloat a) => a -> String
2  bmiTell bmi
3      | bmi <= 18.5 = "You're underweight"
4      | bmi <= 25.0 = "You're supposedly normal"
5      | bmi <= 30.0 = "You're fat!"
6      | otherwise   = "You're a whale"
```

El comando “|” se usa para separar los casos y evitar la repetición de código sobre el nombre de la función.

Sintaxis en funciones

Guards

El comando opcional “where” permite la declaración de variables que pueden ser usadas en todo el cuerpo de la función sin importar el caso, con el objetivo de evitar la repetición de expresión que se usan de manera continua durante la selección de un caso o declarar constantes.

```
1  bmiTell :: (RealFloat a) => a -> a -> String
2  bmiTell weight height
3      | bmi <= skinny = "You're underweight"
4      | bmi <= normal = "You're supposedly normal"
5      | bmi <= fat    = "You're fat!"
6      | otherwise     = "You're a whale"
7      where bmi = weight / height ^ 2
8            skinny = 18.5
9            normal = 25.0
10           fat = 30.0
```

Sintaxis en funciones

Let..in

De manera análoga al “where” usado para crear variables y usarlas en el guard, existe el Let..in, no obstante este aplica de forma más general sobre cualquier expresión, a diferencia del “where” que solo es usado en el mismo guard. La estructura general del Let.. in es:

```
let <bindings> in <expression>.
```

Aplicado sobre una función estándar para hallar el área de un cilindro.

```
1  cylinder :: (RealFloat a) => a -> a -> a
2  cylinder r h =
3      let sideArea = 2 * pi * r * h
4          topArea = pi * r ^2
5      in  sideArea + 2 * topArea
```

Sintaxis en funciones

Let..in

El Let..in puede usar en expresiones simples, como realizar operaciones rápidas sobre una tupla

```
1  (let (a,b,c) = (1,2,3) in a+b+c) * 100
```

E incluso para la compresión de una lista.

```
1  calcBmis :: (RealFloat a) => [(a, a)] -> [a]
2  calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

Recursión

Hallando el máximo elemento de una lista

Para poder implementar esta función utilizaremos algunos de los conceptos mencionados anteriormente.

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

¿Por qué *(Ord a)*?

Como vimos anteriormente se trata de un tipo de

dato que puede ser ordenado, es decir, comparable.

Como ésta función necesita comparar cada uno de los elementos para encontrar el máximo, es necesario que el tipo de dato a se restrinja a la clase ***Ord***.

Recursión

Un ejemplo nunca antes visto: Fibonacci

```
fibo :: ( Integral a ) => a -> a
fibo 0 = 1
fibo 1 = 1
fibo n = fibo( n - 1 ) + fibo( n - 2 )
```

Como bien sabemos el n-ésimo valor de la función Fibonacci depende de sus dos anteriores exceptuando cuando ***n*** vale 0 ó 1.

Usando el esquema de patrones definimos que cuando ***n sea 0*** la función debe retornar 1, y de igual modo en caso de que ***n sea 1***.

Recursión

Replicando un elemento en una lista

```haskell
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

replicate es una función incluida en la librería de Haskell, es una función que *replica* un elemento y devuelve una lista con el elemento especificado replicado un determinado número de veces.

Recursión

Usando el operador “no me importa”

zip es una función incluida en la librería de Haskell, quienes han trabajado en Python probablemente la conozcan.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

Esta función se encarga de tomar dos listas y emparejar un elemento ***x*** de una lista ***p*** con un elemento ***y*** de una lista ***q***, es decir, un subconjunto del producto cruz.

Recursión

Quick sort!

La idea básica de ***quick sort*** consiste en subdividir el problema de ordenamiento en 2 partes (no necesariamente iguales), donde el elemento que divide a la lista se denomina ***pivote***.

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Mediante el uso de ***listas por comprensión*** ponemos a la izquierda del pivote los elementos más pequeños o iguales que él y a la derecha los elementos estrictamente mayores.

Funciones de orden superior

¿Qué son?

La funciones en Haskell pueden retornar funciones.

Cada una de las funciones que cumple esta propiedad se denomina *función de orden superior*.

```
Prelude> max 4 5
5
Prelude> ( max 4 ) 5
5
```

Oficialmente, en Haskell todas la funciones reciben únicamente 1 parámetro.

Funciones de orden superior

Desenvolviendo una simple función...

Veamos la función *multThree*

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

Como podemos ver esta función recibe en teoría 3 parámetros (a) y retorna uno del mismo tipo.

Si son parámetros, ¿por qué no se separan por comas?

En Haskell el operador **->** significa ***retorno***.

Funciones de orden superior

Desde otra perspectiva...

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

- En un script de Haskell (.hs), es válido definir la función de esta manera.
- ¿Cuál es la diferencia? Ninguna.
- El operador -> es asociativo por la derecha!

Funciones de orden superior

Haskell en acción

```
Prelude> let multTwoWithNine = multThree 9
Prelude> multTwoWithNine 2 3
54
Prelude> let multWithEighteen = multTwoWithNine 2
Prelude> multWithEighteen 10
180
```

multThree recibió sólo un parámetro, ¿En serio compila?

Como se mencionó anteriormente, al recibir un sólo parámetro se retornó una función que recibe 2 parámetros, pues ya se “procesó” uno.

Funciones de orden superior

Modificando la asociatividad del operador ->

```haskell
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

```
Prelude> applyTwice ( +3 ) 10
16
Prelude> applyTwice ( ++ " HAHA" ) "HEY"
"HEY HAHA HAHA"
Prelude> applyTwice ( " HAHA" ++ ) "HEY"
" HAHA HAHAHEY"
Prelude> applyTwice ( +3 ) 10
16
Prelude> applyTwice ( ++ " HAHA" ) "HEY"
"HEY HAHA HAHA"
Prelude> applyTwice ( "HAHA " ++ ) "HEY"
"HAHA HAHA HEY"
Prelude> applyTwice ( multThree 2 2 ) 9
144
Prelude> applyTwice ( 3 : ) [ 1 ]
[3,3,1]
```

Funciones de orden superior

Un último ejemplo

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Esta función permite generar una lista a partir de dos listas.

¿Para qué es la función? Al igual que la función ***zip*** se da un emparejamiento, pero la función ***zipWith'*** no generará una tupla, aplicará la función ***f*** a ambos datos y la salida será agregada a la lista retornada.

Funciones de orden superior

Funciones Lambda

Las funciones lambda son básicamente funciones anónimas, se usan en caso de que necesitemos una función que utilizaremos una vez nada más.

Por ejemplo

```
Prelude> zipWith (\a b -> a + 2 * b ) [ 1, 2, 3 ] [ 4, 5, 6 ]
[9,12,15]
```

No todos los días se usa una función que reciba dos parámetros y retorne la suma del primero más el doble del segundo.

Para indicar que se definirá una función lambda se usa el símbolo \ seguido de los parámetros que usará la función.

Funciones de orden superior

2 en 1 : Aplicación y composición de funciones

Aplicación:

Es frecuente que queramos usar el resultado de una función como parámetro para otra función, esta es una forma de hacerlo:

```
Prelude> sum ( map sqrt [ 1 .. 130 ] )
993.6486803921487
```

Esta es equivalente, pero evitamos los paréntesis, yay!

```
Prelude> sum $ map sqrt [ 1 .. 130 ]
993.6486803921487
```

Composición:

Con la misma justificación que la aplicación de funciones, tenemos un caso como el siguiente:

```
Prelude> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Pero es mucho más legible y práctico así

```
Prelude> map ( negate . abs ) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

¡Gracias!

Referencias

- The Haskell Programming Language Wiki,https://wiki.haskell.org/Introduction
- Learn You a Haskell for Great Good! http://learnyouahaskell.com/chapters