UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

Paradigmas de Programación
Paradigma Funcional

Dr. Pablo Vidal

Facultad de Ingenieria
Universidad Nacional de Cuyo

octubre 2020

Tabla de Contenidos

Def. de funciones

Definiciones con condicionales

- Calcular el valor absoluto (con condicionales):

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

- Calcular el signo de un número (con condicionales anidados):

```
signum :: Int -> Int
signum n = if n < 0 then (-1) else
                         if n == 0 then 0 else 1
```

Definiciones con guardas

- Calcular el valor absoluto:

```
abs n | n >= 0 = n
      | otherwise = -n
```

- Calcular el signo de un número:

```
signum n | n < 0 = -1
         | n == 0 = 0
         | otherwise = 1
```

Definiciones con equiparación de patrones: Constantes

- Calcular la negación:

```
not :: Bool -> Bool
not True = False
not False = True
```

- Calcular la conjunción (con valores):

```
(&&) :: Bool -> Bool -> Bool
True && True = True
True && False = False
False && True = False
False && False = False
```

Definiciones con equiparación de patrones: Variables

- Calcular la conjunción (con variables anónimas):

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_ && _ = False
```

- Calcular la conjunción (con variables):

```
(&&) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
```

Definiciones con equiparación de patrones: Tuplas

- Calcular el primer elemento de un par:

```
fst :: (a,b) -> a
fst (x,_) = x
```

- Calcular el segundo elemento de un par:

```
snd :: (a,b) -> b
snd (_,y) = y
```

Alcance Léxico

Las variables locales se pueden crear mediante contructor **let** para proporcionar un alcance dentro de la función. Ejemplo:

Let

```
let two = 2; three = 3 in two * three
```

where

Una instrucción opuesta funciona cuand ose necesita generar binding para trabajar en las guardas:

```
f x y | x>z = ...
      | y==z = ...
      | y<z = ...
      where z=x*x
```

let

```
let <definición> in <expresión>.
```

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

```
ghci>[let sqre x = x * x in (sqre 5, sqre 3, sqre 2)]
[(25,9,4)]
```

Nota

No pueden ser usadas entre guardas.

where

Algunos prefieren where porque las variables vienen después de la función que los utiliza. De esta forma, el cuerpo de la función esta más cerca de su nombre y declaración de tipo y algunos piensan que es más legible.

```
raices :: (Float, Float, Float) -> (Float, Float)
raices (a,b,c) = (x1, x2) where
   x1 = e + sqrt d / (2 * a)
   x2 = e - sqrt d / (2 * a)
   d = b * b - 4 * a * c
   e = - b / (2 * a)
```

Condicionales

If

```
if e1 then e2 else e3
```

Case

```
case e1 of True -> e2
           False -> e3
```

Definiciones con equiparación de patrones: Listas

(test1 xs) se verifica si xs es una lista de 3 caracteres que empieza por 'a'.

```
test1 :: [Char ] -> Bool
test1 ['a',_,_] = True
test1 _ = False
```

Construcción de listas con (:)

```
[1,2,3] = 1:[2,3] = 1:(2:[3]) = 1:(2:(3:[]))
```

(test2 xs) se verifica si xs es una lista de caracteres que empieza por 'a'.

```
test2 :: [Char ] -> Bool
test2 ('a':_) = True
test2 _ = False
```

Decidir si una lista es vacía:

```
null :: [a] -> Bool
null [] = True
null (_:_) = False
```

Primer elemento de una lista:

```
head :: [a] -> a
head (x:_) = x
```

Resto de una lista:

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

Registro

```
data Coordenada = Plano Float Float |
                  Espacial Float Float Float
                  deriving (Eq, Ord, Show)
```

Ejemplo con definición de tipos - Persona

```
data Persona = Persona String String Int Float String
               deriving (Show)

ghci> let diva = Persona "Lady" "Gaga" 32 52.0  "Cantar"
ghci> diva
Persona "Lady" "Gaga" 32 52.0 "Chocolate"
```

Ejemplo con definición de tipos - Persona

```haskell
primerNombre :: Persona -> String
primerNombre (Persona fn _ _ _ _) = fn

apellido :: Persona -> String
apellido (Persona _ ln _ _ _) = ln

edad :: Persona -> Int
edad (Persona _ _ ed _ _) = ed

peso :: Persona -> Float
peso (Persona _ _ _ we _) = we


habilidad :: Persona -> String
habilidad (Persona              sk) = sk
```

Ejemplo con definición de tipos - Persona

```
ghci> let diva = Persona "Lady" "Gaga" 32 52.0  "Cantar"
ghci> diva
Persona "Lady" "Gaga" 32 52.0 "Chocolate"
ghci> apellido diva
"Lady"
ghci> peso diva
52.0
ghci> habilidad diva
"Cantar"
```

Registros

```
data Persona = Persona { primerNombre :: String
                       , apellido :: String
                       , edad :: Int
                       , peso :: Float
                       , habilidad :: String
                       } deriving (Show)
```

```
ghci> :t habilidad
habilidad :: Persona -> String
ghci> :t primerNombre
primerNombre :: Persona -> String
```

Expresiones Lambda

Expresiones lambda

- Las funciones pueden construirse sin nombrarlas mediante las expresiones lambda.
- Ejemplo de evaluación de expresiones lambda:

```
ghci> (\x -> x+x) 3
6
```

Expresiones lambda y parcialización

Uso de las expresiones lambda para resaltar la parcialización: I (suma x y) es la suma de x e y. I Definición sin lambda:

```
suma x y = x+y
I Definición con lambda:
suma' = \x -> (\y -> x+y)
```

Expresiones lambda y funciones como resultados

Uso de las expresiones lambda en funciones como resultados:
(const x y) es x. Definición sin lambda:

```
const :: a -> b -> a
const x = x
```

Definición con lambda:

```
const' :: a -> (b -> a)
const' x = \_ -> x
```

Expresiones lambda y funciones de sólo un uso

Uso de las expresiones lambda en funciones con sólo un uso:

- (impares n) es la lista de los n primeros números impares.
- Definición sin lambda:

```
impares n = map f [0..n-1]
            where f x = 2*x+1
```

- Definición con lambda:

```
impares' n = map (\x -> 2*x+1) [0..n-1]
```

Porque usar expresiones lambda

- Podemos crear código más claro y conciso, ya que las expresiones lambda nos permiten referenciar métodos anónimos o métodos sin nombre

Funciones recursivas

Recursión numérica: El factorial

- La función factorial:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- Cálculo:

```
factorial 3 = 3 * (factorial 2)
= 3 * (2 * (factorial 1))
= 3 * (2 * (1 * (factorial 0)))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

Producto de una lista de números:

```
product :: Num a => [a] -> a
product [] = 1
product (n:ns) = n * product ns
```

Cálculo:

```
product [7,5,2] = 7 * (product [5,2])
= 7 * (5 * (product [2]))
= 7 * (5 * (2 * (product [])))
= 7 * (5 * (2 * 1))
= 7 * (5 * 2)
= 7 * 10
= 70
```

Recursión sobre listas: La función length

Longitud de una lista:

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

Cálculo:

```
length [2,3,5] = 1 + (length [3,5])
= 1 + (1 + (length [5]))
= 1 + (1 + (1 + (length [])))
= 1 + (1 + (1 + 0))
= 1 + (1 + 1)
= 1 + 2
= 3
```

Recursión sobre listas: La función reverse

Inversa de una lista:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Cálculo:

```
reverse [2,5,3] = (reverse [5,3]) ++ [2]
= ((reverse [3]) ++ [5]) ++ [2]
= (((reverse []) ++ [3]) ++ [5]) ++ [2]
= (([] ++ [3]) ++ [5]) ++ [2]
= ([3] ++ [5]) ++ [2]
= [3,5] ++ [2]
= [3,5,2]
```

Recursión sobre listas: ++

Concatenación de listas:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Cálculo:

```
[1,3,5] ++ [2,4] = 1:([3,5] ++ [2,4])
= 1:(3:([5] ++ [2,4]))
= 1:(3:(5:([] ++ [2,4])))
= 1:(3:(5:[2,4]))
= 1:(3:[5,2,4])
= 1:[3,5,2,4]
= [1,3,5,2,4]
```

Recursión sobre listas: Inserción ordenada

(*inserta el xs*) inserta el elemento *el* en la lista *xs* delante del primer elemento de *xs* mayor o igual que *el*. Por ejemplo,

```
inserta 5 [2,4,7,3,6,8,10]   [2,4,5,7,3,6,8,10]
inserta :: Ord a => a -> [a] -> [a]
inserta e [] = [e]
inserta e (x:xs) | e <= x = e : (x:xs)
                 | otherwise = x : inserta e xs
```

Cálculo:

```
inserta 4 [1,3,5,7] = 1:(inserta 4 [3,5,7])
= 1:(3:(inserta 4 [5,7]))
= 1:(3:(4:(5:[7])))
= 1:(3:(4:[5,7]))
= [1,3,4,5,7]
```

Recursión sobre listas: Ordenación por inserción

(ordena_por_insercion xs) es la lista xs ordenada mediante inserción, Por ejemplo,

```
ordena_por_insercion [2,4,3,6,3]    [2,3,3,4,6]
ordena_por_insercion :: Ord a => [a] -> [a]
ordena_por_insercion [] = []
ordena_por_insercion (x:xs) =
inserta x (ordena_por_insercion xs)
```

Cálculo:

```
ordena_por_insercion [7,9,6] =
= inserta 7 (inserta 9 (inserta 6 []))
= inserta 7 (inserta 9 [6])
= inserta 7 [6,9]
= [6,7,9]
```

Recursión sobre varios argumentos: La función zip

Emparejamiento de elementos (la función zip):

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

Cálculo:

```
zip [1,3,5] [2,4,6,8]
= (1,2) : (zip [3,5] [4,6,8])
= (1,2) : ((3,4) : (zip [5] [6,8]))
= (1,2) : ((3,4) : ((5,6) : (zip [] [8])))
= (1,2) : ((3,4) : ((5,6) : []))
= [(1,2),(3,4),(5,6)]
```

Recursión sobre varios argumentos: La función drop

Eliminación de elementos iniciales:

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

Cálculo:

```
drop 2 [5,7,9,4] | drop 5 [1,4]
= drop 1 [7,9,4] | = drop 4 [4]
= drop 0 [9,4] | = drop 1 []
= [9,4] | = []
```

Recursión mutua: Par e impar

Par e impar por recursión mutua:

```
par :: Int -> Bool
par 0 = True
par n = impar (n-1)
impar :: Int -> Bool
impar 0 = False
impar n = par (n-1)
```

Cálculo:

```
impar 3 | par 3
= par 2 | = impar 2
= impar 1 | = par 1
= par 0 | = impar 0
= True | = False
```

Recursión mutua: Posiciones pares e impares

- (pares xs) son los elementos de xs que ocupan posiciones pares.
- (impares xs) son los elementos de xs que ocupan posiciones impares.

```
pares :: [a] -> [a]
pares [] = []
pares (x:xs) = x : impares xs
impares :: [a] -> [a]
impares [] = []
impares (_:xs) = pares xs
```

Cálculo:

```
pares [1,3,5,7]
= 1:(impares [3,5,7])
= 1:(pares [5,7])
= 1:(5:(impares [7]))
= 1:(5:[])
= [1,5]
```

Composiciones de funciones

Definiciones por composición

- Decidir si un carácter es un dígito:

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

- Decidir si un entero es par:

```
even :: (Integral a) => a -> Bool
even n = n `rem` 2 == 0
```

- Dividir una lista en su n-ésimo elemento:

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

Composición de funciones

- Las funciones son **chiquitas** y **cohesivas**.
- Las combinamos entre ellas para construir algoritmos más grandes.

Es el proceso de usar la salida de una función como entrada de otra función.
En matemáticas, la composición se denota por f g (x) donde g() es una función y su salida se usa como entrada de otra función, es decir, f().

Composición de funciones

```
eveno :: Int -> Bool
noto  :: Bool -> String

eveno x = if x `rem` 2 == 0 then True
          else False
noto x = if x == True then "This is an even Number"
         else "This is an ODD number"

main = print ((noto.eveno)(16))
```

En la función principal, estamos llamando a dos funciones, *noto* y *eveno*, simultáneamente. El compilador primero llamará a la función *eveno ()* con 16 como argumento. A partir de entonces, el compilador utilizará la salida del método *eveno* como entrada del método *noto ()*.

Composición de funciones

Escenarios donde esto es especialmente útil, es eliminar la necesidad de una lambda explícita. P.ej:

```
map (\x -> not (even x)) [1..9]
```

puede ser reescrito como:

```
map (not . even) [1..9]
```

Composición de funciones

Este ejemplo es artificial, pero supongamos que tenemos

```
sqr x = x * x
inc x = x + 1
```

y queremos escribir una función que calcule `x^2+1`. Podemos escribir

```
xSquaredPlusOne = inc . sqr
```

lo que significa

```
xSquaredPlusOne x = (inc . sqr) x
```

lo que significa

```
xSquaredPlusOne x = inc(sqr x)
```

dado que f=inc y g=sqr.

Composición de funciones

- **desort** es una función que ordena una lista a la inversa.
- Básicamente, *desort* alimenta sus argumentos en *sort*, y luego alimenta el valor de retorno de sort en reverse, y devuelve eso. Entonces lo ordena y luego invierte la lista ordenada.

```
desort = (reverse . sort)

reverse :: [a] -> [a]
sort :: Ord a => [a] -> [a]
```

Listas por comprensión

Listas por comprensión

- Suponen un modo más elegante y rápido de escribir código, sino que además se ejecutan más rápidamente también.
- Las listas por comprensión debutaron por vez primera en un lenguaje de programación en 1977, cuando Rod Burstall y John Darlington diseñaron el lenguaje funcional NPL.
- Hoy en día un número importante de lenguajes de programación soportan listas por comprensión, incluyendo: Haskell, JavaScript, CoffeeScript, Erlang, F#, Scala, Clojure y Racket, entre otros.

Listas por comprensión

- ¿Qué es un conjunto?
- ¿Cómo se define un conjunto?

Listas por comprensión

- En matemáticas, un conjunto es simplemente una colección de elementos bien definidos. Los elementos pueden ser cualquier cosa: números, letras, nombres, figuras, etc.

Listas por extensión y comprensión

- Hay dos manera de definir los elementos que pertenecen a un conjunto: por extensión o por comprensión. Cuando se define un conjunto por extensión cada elemento se enumera de manera explícita.
- Tomando un ejemplo inspirado en el Señor de los Anillos de J. R. R. Tolkien:
  - A = { Frodo, Sam, Pippin, Merry, Legolas, Gimli, Aragorn, Boromir, Gandalf }
- La expresión anterior se lee así: **A es el conjunto formado por los elementos: Frodo, Sam, Pippin, Merry, Legolas, Gimli, Aragorn, Boromir y Gandalf**.

Listas por comprensión

- Por otro lado, cuando un conjunto se define por comprensión no se mencionan los elementos uno por uno sino que se indica una propiedad que todos éstos cumplen, por ejemplo:
- $\mathrm{B} = \{x | \, x \in \text{Comunidad del Anillo}\}$
- La expresión de arriba se lee así: **B es el conjunto de elementos x tales que x pertenece ($\in$) a la *Comunidad del Anillo*.**

Listas por comprensión

- En los ejemplos anteriores podemos decir que el conjunto A es igual al conjunto B debido a que ambos tienen exactamente los mismos elementos.
- Esto es cierto aún a pesar de que el conjunto A se definió por **extensión** y B se definió por **comprensión**.

Definiciones por comprensión

- Definiciones por comprensión en Matemáticas:
  - $\{x^2 : \text{x } \{2, 3, 4, 5\}\} = \{4, 9, 16, 25\}$
- Definiciones por comprensión en Haskell:

```
ghci> [x^2 | x <- [2..5]]
[4,9,16,25]
```

- La expresión x <- [2..5] se llama un generador.
- Ejemplos con más de un generador:

```
ghci> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
ghci> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Generadores dependientes

- Ejemplo con generadores dependientes:

```
ghci> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

- (concat xss) es la concatenación de la lista de listas xss. Por ejemplo,

```
concat [[1,3],[2,5,6],[4,7]]    [1,3,2,5,6,4,7]
```

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

Generadores con variables anónimas

- Ejemplo de generador con variable anónima: (primeros ps) es la lista de los primeros elementos de la lista de pares ps. Por ejemplo,

```
primeros [(1,3),(2,5),(6,3)]    [1,2,6]
primeros :: [(a, b)] -> [a]
primeros ps = [x | (x,_) <- ps]
```

- Definición de la longitud por comprensión

```
length :: [a] -> Int
length xs = sum [1 | _ <- xs]
```

Guardas

- Las listas por comprensión pueden tener guardas para restringir los valores.
- Ejemplo de guarda:

```
ghci> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

  La guarda es even x.
- (factores n) es la lista de los factores del número n. Por ejemplo,

```
factores 30   [1,2,3,5,6,10,15,30]
factores :: Int -> [Int]
factores n = [x | x <- [1..n], n `mod` x == 0]
```

Guardas: Cálculo de primos

- (primo n) se verifica si n es primo. Por ejemplo,

```
primo 30   False
primo 31   True
primo :: Int -> Bool
primo n = factores n == [1, n]
```

- (primos n) es la lista de los primos menores o iguales que n. Por ejemplo:

```
primos 31   [2,3,5,7,11,13,17,19,23,29,31]
primos :: Int -> [Int]
primos n = [x | x <- [2..n], primo x]
```

Guarda con igualdad

- Una lista de asociación es una lista de pares formado por una clave y un valor. Por ejemplo,

  ```
  [("Juan",7),("Ana",9),("Eva",3)]
  ```

- (busca c t) es la lista de los valores de la lista de asociación t cuyas claves valen c. Por ejemplo,

  ```
  ghci> busca 'b' [('a',1),('b',3),('c',5),('b',2)]
  [3,2]
  busca :: Eq a => a -> [(a, b)] -> [b]
  busca c t = [v | (c', v) <- t, c' == c]
  ```

Cadenas y listas

- Las cadenas son listas de caracteres. Por ejemplo,

```
ghci> "abc" == ['a','b','c']
True
```

- La expresión

```
"abc" :: String
```

es equivalente a

```
['a','b','c'] :: [Char]
```

- Las funciones sobre listas se aplican a las cadenas:

```
length "abcde"   5
reverse "abcde"   "edcba"
"abcde" ++ "fg"   "abcdefg"
posiciones 'a' "Salamanca"   [1,3,5,8]
```

Definiciones sobre cadenas con comprensión

- (minusculas c) es la cadena formada por las letras minúsculas de la cadena c. Por ejemplo,

```
minusculas "EstoEsUnaPrueba"   "stosnarueba"
minusculas :: String -> String
minusculas xs = [x | x <- xs, elem x ['a'..'z']]
```

- (ocurrencias x xs) es el número de veces que ocurre el carácter x en la cadena xs. Por ejemplo,

```
ocurrencias 'a' "Salamanca"   4
ocurrencias :: Char -> String -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']
```

Referencias


R. Bird. , 2000, ***Introducción a la programación funcional con Haskell***, Prentice Hall.


Graham Hutton, 2007, ***Programming in Haskell***, Cambridge University Press, New York, NY, USA.


O'Sullivan, Bryan, Stewart, Don and Goerzen, John, 2008, ***Real World Haskell***, O'Reilly.


B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo, 2004, ***Razonando con Haskell***, Thompson.


S. Thompson , 1999, ***Haskell: The Craft of Functional Programming, Second Edition***, Addison-Wesley.