**Clase 09-04-2021**

Programa:

* Secuencia de pasos/instrucciones
* Variables (posiciones de memoria que guardan valores intermedios mediante la asignación)
* Estructuras de datos (listas, árboles)
* Estructuras de control (secuencia, selección, iteración)
* Estado final (se debe verificar que sea correcto)

Paradigma:

* Def 1: modelo o marco teórico en el cual uno se basa para resolver un problema
* Def. 2: conjunto de herramientas conceptuales que configuran una respuesta a la pregunta de qué es un programa

Clasificación de paradigmas:

* Procedural o Imperativo:
  + Las instrucciones se ejecutan en un orden determinado
  + Se decide cómo resolver el problema
  + Se tiene control sobre el algoritmo
  + La asignación destructiva solo se da en este paradigma, no en el declarativo.
* Declarativo:
  + Se describen las características del problema
  + Hay un “motor” externo que se encarga de encontrar la solución
  + Hay menos control sobre el algoritmo
* Es importante crear abstracciones auxiliares para separar las ideas del problema y concentrarse en lo importante.
* Al separar el problema en partes, se puede reutilizar código.
* Es importante mejorar la expresividad con estas abstracciones, por ejemplo, usando palabras entendibles.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Clase 16-04-2021**

Función:

Definición: una **transformación** de una entrada en una salida.

* Es la abstracción fundamental del paradigma funcional.
* En Haskell se describen mediante ecuaciones.
* Cada vez que apliquemos una función a un valor siempre vamos a obtener el mismo resultado.
* Siempre vamos a tener Transparencia Referencial.
* Puedo reemplazar una expresión cualquiera por el resultado y todo sigue funcionando.

Variable:

* En el paradigma imperativo es un espacio de memoria donde almaceno valores intermedios.
* En el paradigma funcional es una incógnita, un valor que no fue aún calculado.

Tuplas:

Permiten representar un tipo de dato compuesto, pero con elementos que pueden ser de distinto tipo. El número de elementos es fijo (siempre el mismo).

* Tuplas de dos elementos: (5,4), (True, 7)
* Tuplas de tres elementos: (“Hola”, 14, False), (23, 04, 2021)
* Usar ecuaciones con guardas hace que nuestras funciones sean más imperativas. Al no usar guardas hacemos que nuestro código sea más declarativo.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Clase 23-04-2021**

Vemos el laboratorio de funciones: <https://uqbar-project.github.io/function-laboratory/>

* “$” (Apply): Aplica un parámetro a una función.

Ej. not.even $ 5 Le aplico el parámetro 5 a una función.

Currificación**:**

Todas las funciones reciben un único parámetro y devuelven un resultado.

Por ejemplo, si tenemos:

suma :: Integer -> Integer -> Integer

suma x y = x + y

Quedaría, usando Currificación, de la siguiente forma:

suma :: Integer -> (Integer -> Integer)

(suma x) y = x + y

Aplicación Parcial:

Al estar currificada, una función se puede aplicar parcialmente. O sea, la “dejamos lista” para que pida un solo parámetro y devuelva un resultado.

suma 10 es una función que surge de la aplicación parcial de suma ya que necesita un solo parámetro para devolver un resultado.

* import Text.Show.Functions Permite imprimir las funciones en pantalla.

Prelude> suma 10

<function>

Composición de funciones:

f (g(x)) en Haskell es: f.g

Ej. esParSiguiente :: Integer -> Bool

esParSiguiente x = (even.siguiente) x

Expresiones Lambda – Funciones Anónimas:

Son funciones que no tienen definición, no tienen. Sirven cuando a una función le quiero pasar por parámetro otra función.

\param1 -> \param2 … -> body

O

\param1 param2 … -> body

**Modelado de información**

Tipos de datos simple:

Aquellos datos que no se pueden descomponer:

* 2 :: Integer
* 2.6 :: Double
* True :: Bool
* ‘z’ :: Char

Tipos de datos compuestos:

Se pueden descomponer:

* “Hola Mundo” :: String
* [‘5’, ‘c’, ‘j’, ‘7’] :: [Char]
* (‘a’, False) :: (Char, Bool)
* doble :: Integer -> Integer
* [3,4 5,6] :: [Integer]
* Estudiante “Ana” [8,10, 9] :: Persona (\*)

Listas:

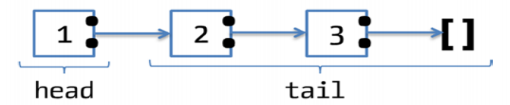
Conjunto de elementos del mismo tipo. Por ejemplo:

* Lista de booleanos: [True, False] :: [Bool]
* Una lista vacía es de tipo genérico: [] :: [a]

Construcción de una lista:

[1, 2, 3] se puede descomponer en (1:[2, 3]) o (1:2:3:[])

* Puedo tomar cuantas head como quiera.
* La cola, siempre es una lista, que puede estar vacía.



Pattern Matching:

Mecanismo interno que permite verificar si un patrón coincide con un valor.

Ej.

Pattern 🡪 (*h:x:t*) es una lista separada en cabeza y cola.

* Si pongo [1,2
* ] 🡪 *h* es 1, x es 2, t es []
* Si pongo [1,2,3,4] 🡪 *h* es 1, x es 2, *t* es [3,4]
* Si pongo [1] 🡪 No hay pattern matching

Pattern 🡪 [x, y] es una lista de solo dos elementos.

* Si pongo [1,2] 🡪 *x* es 1, *y* es 2
* Si pongo [1,2,3] 🡪 No hay pattern matching

Funciones para el manejo de listas:

* head (cabeza de la lista)
* tail (cola de la lista)
* length (cantidad de elementos de una lista)
* ++ (concatena listas del mismo tipo)
* take n (primeros n elementos de la lista)

Tuplas:

* Son un tipo de dato compuesto.
* Cada elemento puede ser de distinto tipo.
* La cantidad de elementos es fija.

Prelude>:t (‘a’, False)

(Char, Bool)

En tuplas de dos elementos, puedo usar *fst* y *snd* para obtener el primero o el segundo elemento de la lista.

Listas vs. Tuplas:

Listas:

* Tipo de dato recursivo (porque la cola es una lista).
* Cantidad de elementos variables.
* Elementos homogéneos (del mismo tipo).

Tuplas:

* No recursivas
* Cantidad de elementos fija.
* Elementos heterogéneos (distintos tipos).
* Tanto las tuplas como las listas son tipos de datos compuestos que se pueden descomponer con pattern matching.
* Para crear una lista infinita se debe escribir “..”. Por ejemplo, “[15..]” me genera una lista de 15 a infinito.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Clase 30-04-2021**

**Modelado de información**

Tipo de datos:

* Conjunto de elementos con características en común.
* Conjunto de operaciones para manipular dichos elementos.

Para definir un tipo de dato hay que:

* Establecer la forma de cada elemento.
* Establecer un mecanismo de acceso a cada elemento.

Tipos algebraicos:

* Tipos Propios
* Tuplas

Constructores:

* Comienzan con mayúscula.
* No tienen asociada una regla de reducción.
* Puede o no tener argumentos.

data:

* Introduce un nuevo tipo algebraico
* Introduce los nombres de los constructores.
* Define los tipos de los argumentos de los constructores

data ColoresPrimarios = Rojo|Amarillo|Azul

data Figura = Circulo (radio :: Double) | Rectangulo (base :: Double, altura :: Double)

Pattern Matching:

Pattern (expresión especial):

* Sólo con constructores y variables.
* Argumento en el lado izquierdo de una ecuación.

Matching (operación asociada a un pattern)

* Inspecciona el valor de una expresión.
* Puede fallar o tener éxito.
* Si tiene éxito, liga las variables del pattern.

data Punto = Plano {coorX :: Double, coorY :: Double} deriving Show|

Espacio {coorX :: Double, coorY :: Double, coorZ :: Double} deriving Show

distancia :: Punto -> Double

distancia (Plano x y) = sqrt (x^2 + y^2)

distancia (Espacio x y z) = sqrt (x^2 + y^2 +z^2)

* “Punto” es un tipo propio con dos constructores, “Plano” y “Espacio”.
* “Plano tiene dos argumentos, “Espacio” tiene tres.
* distancia es una función que espera un “Punto” y devuelve un “Double”.
* “deriving Show” establece que el tipo de dato se puede mostrar por pantalla.

Tipos propios vs. Tuplas:

* Una tupla NO es intuitiva.
* Los tipos propios son más expresivos.
* Los tipos propios son extensibles.

Conviene usar tipos propios.

**Listas por comprensión – Orden superior**

Listas por comprensión:

* Notación especial para escribir listas.
* Similar a la notación de conjuntos por comprensión.
* El orden importa.

Ej. numerosPares numeros = [num | num <- numeros, even num]  
(Devuelve los números pares del conjunto que le pase por parámetro)

Ej. Prelude> [ x^2 | x <- [1..10], even x ]  
(Devuelve el cuadrado de los numeros pares)

Ej. Prelude> [(i,j) | i <- [1..3], j <- ['a','b']]  
(Devuelve el producto cartesiano)

Función de Orden Superior:

Una función es de orden superior si recibe como argumento o devuelve una función

Ej. seleccionar :: (Integer -> Bool) -> [Integer] -> [Integer]

seleccionar criterio numeros = [num | num <- números, criterios num]

“seleccionar” recibe una función (criterio de selección) y un conjunto de números, y devuelve el conjunto de números con el criterio de selección aplicado.

Invocar una función de orden superior:

Main> seleccionar even [2..10]

[2,4,6,8,10]

Main> seleccionar (esDivisiblePor 3) [2..10]

[3,6,9]

Main> seleccionar (>5) [2..10]

[6,7,8,9,10]

Funciones predefinidas de orden superior:

* filter: selecciona a los elementos de la lista que cumplan con una condición determinada.

filter :: (a -> Bool) -> [a] -> [a]

filter criterio lista = [elemento | elemento <- lista, criterio elemento]  
-----------

Prelude> filter (even.fst) [(2,5), (5, 8), (4,6)]

[(2,5), (4, 6)]

* map: transforma una lista de elementos en otra, aplicando la función que recibe por parámetro a todos los elementos.

map :: (a -> b) -> [a] -> [b]

map criterio lista = [ criterio elemento | elemento <- lista]  
-----------

Prelude> map snd [(2,5), (5, 8), (4,6)]

[5, 8, 6]

* any: indica si alguno de los elementos de la lista cumple una condición determinada.

any :: (a -> Bool) -> [a] -> Bool  
---------

Prelude> any (elem 2) [[3,5,6], [7,2,9]]

True

* all: indica si todos los elementos de la lista cumplen una condición determinada.

all :: (a -> Bool) -> [a] -> Bool  
----------

Prelude> all ((>4).length) [“emilce”, “julio”, “marisa”]

True

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Clase 07-05-2021**

**Recursividad y Orden Superior:**

La recursividad viene de la mano de la inducción, que nos permite:

* Definir conjuntos infinitos.
* Definir funciones recursivas sobre ellos, con garantía de terminación.
* Probar propiedades sobre sus elementos.

Si una función está definida recursivamente, entonces, se auto invoca.

Ej. Factorial:

*n! = 1 × 2 × 3 × ... × (n − 1) × n*

Factorial en Haskell:

factorial 0 = 1 caso base

factorial n = n \* factorial (n – 1) caso inductivo

Recursividad con listas:

Cantidad de elementos de una lista:

length [] = 0 caso base

length (x:xs) = 1 + length xs caso inductivo

Último elemento de una lista:

last [x] = x caso base

last (x:xs) = last xs caso inductivo

Saber si un elemento pertenece a un conjunto:

elem \_ [] = False caso base

elem e (x:xs) = e == x || elem e xs caso inductivo

Invertir el orden de los elementos de una lista:

reverse [] = [] caso base

reverse (x:xs) = (reverse xs) ++ [x] caso inductivo

Máximo de una lista:

maximo [x] = x caso base

maximo (x:xs) = x `max` maximo xs caso inductivo

Funciones de Orden Superior definidas con recursividad:

Función map:

map :: (a -> b) -> [a] -> [b]

map \_ [] = [] caso base

map f (x:xs) caso inductivo

Función filter:

filter :: (a -> Bool) -> [a] -> [a]

filter \_ [] = [] caso base

filter f (x:xs) | f x = x : filter f xs

| otherwise = filter f xs

Función all:

all :: (a -> Bool) -> [a] -> Bool

all \_ [] = True caso base

all f (x:xs) = f x && all f xs caso inductivo

Función any:

any :: (a -> Bool) -> [a] -> Bool

any \_ [] = False caso base

any f (x:xs) = f x || any f xs caso inductivo

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Clase 14-05-2021**

**Familia Fold**

Fold es una función de orden superior que procesa una estructura de datos en un cierto orden y construye un valor de retorno.

Función Foldl:

Foldl (“l” de left) va aplicando la función al valor neutro (semilla) y a los elementos de la lista de izquierda a derecha.

Definición:

foldl :: (a → b → a) → a → [b] → a

foldl \_ valorNeutro [] = valorNeutro

foldl funcion valorNeutro (cabeza:cola) = foldl funcion (funcion valorNeutro cabeza) cola

Función Foldr:

Foldr (“r” de right) va aplicando la función primero al elemento de la lista y luego al valor neutro (semilla), o sea, va de derecha a izquierda. Devuelve un valor del mismo tipo que la semilla.

foldr :: (b → a → a) → a → [b] → a

foldr \_ valorNeutro [] = valorNeutro

foldr funcion valorNeutro (cabeza:cola) = funcion cabeza (foldr funcion valorNeutro cola)

Comparativa con la función (+):

Prelude> foldl (+) 0 [2, 3, 7, 5]

foldl (+) 0 [2, 3, 7, 5]

foldl (+) ((+) 0 2 ) [3, 7, 5]

foldl (+) 2 [3, 7, 5]

foldl (+) ((+) 2 3) [7, 5]

foldl (+) 5 [7, 5]

foldl (+) ((+) 5 7) [5]

foldl (+) 12 [5]

foldl (+) ((+) 12 5) []

foldl (+) 17 []

17

Prelude> foldr (+) 0 [2, 3, 7, 5]

(+) 2 (foldr (+) 0 [3, 7, 5])

(+) 2 ((+) 3 (foldr (+) 0 [7, 5])

(+) 2 ((+) 3 ((+) 7 (foldr (+) 0 [5])

(+) 2 ((+) 3 ((+) 7 ((+) 5 (foldr (+) 0 [])

(+) 2 ((+) 3 ((+) 7 ((+) 5 (foldr (+) 0 [])

(+) 2 ((+) 3 ((+) 7 ((+) 5 0)

17

Comparativa con la función (-):

>foldl (-) 0 [1, 2, 5]

foldl (-) ((-) 0 1) [2, 5]

foldl (-) -1 [2, 5]

foldl (-) ((-) (-1) 2) [5]

foldl (-) -3 [5]

foldl (-) ((-) (-3) 5) []

foldl (-) -8 []

-8

>foldr (-) 0 [1, 2, 5]

(-) 1 (foldr (-) 0 [2, 5])

(-) 1 ((-) 2 (foldr (-) 0 [5]))

(-) 1 ((-) 2 ((-) 5 (foldr (-) 0 [])) )

(-) 1 ((-) 2 ((-) 5 0))

(-) 1 ((-) 2 5)

(-) 1 -3

4

* Con operaciones conmutativas da lo mismo, pero con las no conmutativas NO.

Función foldl1:

Es como foldl, pero toma como valor inicial (semilla) a la cabeza de la lista. No se puede aplicar a listas vacías (ya que no habría cabeza).

foldl1 :: (a -> a -> a) -> [a] -> a

foldl1 f (x:xs) = foldl f x xs

Función foldr1:

Es como foldr, pero toma como valor inicial (semilla) al último elemento de la lista. No se puede aplicar a listas vacías (ya que no habría cabeza).

foldr1 :: (a -> a -> a) -> [a] -> a

foldr1 f [x] = x

foldr1 f (x:xs) = f x (foldr1 f xs)

Ejemplos:

Sumatoria de una lista con foldl1:

sumatoria lista = foldl1 (+) lista

Productoria de una lista con foldl1:

productoria lista = foldl1 (\*) lista

Máximo de una lista con foldl1:

maximo lista = foldl1 max lista

Comparativa entre foldl1 y foldr1:

> foldl1 (-) [2, 4, 6]

-8

> foldl1 div [8,4,2]

1

> foldr1 (-) [2, 4, 6]

4

> foldr1 div [8,4,2]

4

* Con operaciones conmutativas da lo mismo, pero con las no conmutativas NO.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Clase 21-05-2021**

**Estrategias de evaluación y listas infinitas**

Transparencia Referencial:

Característica de las operaciones:

* Independientes
* Determinísticas
* Sin efecto

Se puede reemplazar su invocación por su resultado.

Estrategias de reducción:

Dadas las siguientes funciones:

doble x = x + x

siguiente x = x + 1

doble (siguiente 2)

doble (2 + 1)

doble 3

3 + 3

6

Evaluación ansiosa (**eager**): va de adentro hacia afuera.

doble (siguiente 2)

(siguiente 2)+(siguiente 2)

(2 + 1) + (2 + 1)

3 + 3

6

Evaluación perezosa (**lazy**): va de afuera hacia adentro.

Gracias a que son operaciones con transparencia referencial, de las dos formas dan lo mismo.

Evaluación Lazy:

Evalúa solo lo que necesita (puede ahorrar procesamiento):

> False && (factorial 100000000000000 > 0)

False

> even 2 || elem 0 [1 .. 999999999999999999999999]

True

Haskell usa, por defecto, la evaluación lazy.

Función *take*:

*take* toma una cantidad (que se le pasa por parámetro) de elementos de una lista.

Ej.

> take 10 [20..]

[20,21,22,23,24,25,26,27,28,29]

Números primos:

esDivisorDe :: Int -> Int -> Bool

esDivisorDe multiplo divisor = multiplo `mod` divisor == 0

esPrimo :: Int -> Bool

esPrimo 1 = False

esPrimo n = all (not . esDivisorDe n) [2..(n-1)]

Teniendo:

primos :: [Int]

primos = filter esPrimo [1..]

Podemos sumar los primeros 25 números primos y encontrar el primer primo mayor a 1000:

> sum . take 25 $ primos

1060

> head . filter (> 1000) $ primos

1009

No podemos sumar todos los primos o encontrar el primer primo menor a 0:

> sum primos

NO TERMINA

> head . filter (< 0) $ primos

NO TERMINA

**Sistemas de Tipos en Haskell**

Tipos genéricos:

* Función *id:*

id :: a -> a

id x = x

* Función *const*:

const :: a -> b -> a

const x \_ = x

Polimorfismo paramétrico:

Característica de las funciones que pueden aceptar valores de cualquier tipo, sin ninguna restricción.

Funciones genéricas con tuplas:

* Función *fst*:

fst :: (a,b) -> a

* Funcion *snd*:

snd :: (a,b) -> b

*a* y *b* son variables de tipo diferentes. De esta forma la tupla puede tener dos elementos de distinto tipo.

Funciones genéricas con listas:

* Función *head*:

head :: [a] -> a

* Función *last*:

last :: [a] -> a

* Función *tail*:

tail :: [a] -> [a]

Typeclass:

* Agrupa tipos.
* Define un conjunto de operaciones que todos los tipos que pertenecen a dicho Typeclass deben implementar.

Ej.

Num agrupa los tipos Integer, Float, Int y Double.

Operaciones de tipo Num:

(+),(-),(\*) :: (Num a) => a -> a -> a

negate, abs, signum :: (Num a) => a -> a