

FrecuenciaCardiacaPromedio = 80

frecuenciaCardiacaPromedio = 90

X ↗

rompe cuando cargo el programa



No hay variables, poner algo así es como darle una "etiqueta", un "alias" que le doy a un número

TieneTaquicardia :: Int → Bool

TieneTaquicardia unaFrecuencia = unaFrecuencia ≥ 100

hacerActividad :: Int → Int

hacerActividad unaFrecuencia = unaFrecuencia + 50

> FrecuenciaCardiacaPromedio

80

> TieneTaquicardia freqCardProm

false

> hacerActividad freqCardProm

130

> tieneTaquicardia freqCardProm

False

El 130 no se "guardó"

NO hay cambios en TIEMPOS de ejecución en funcional.

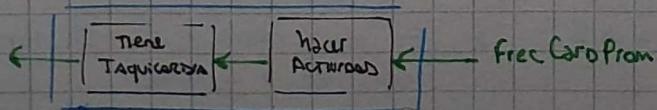
130 es una imagen de la función hacerActividad

Transparencia REFERENCIAL → Lo que tengo a la IZQ y a la DERECHA es lo MISMO. NO hay ASIGNACIÓN, es el IGUAL de MATEMÁTICA.

Composición de Funciones

$$\boxed{\text{TieneTaquicardia} \circ \text{hacerActividad}} \quad \& \quad \text{FrecuenciaCardiacaPromedio}$$

$$\boxed{(\text{TieneTaquicardia} \circ \text{hacerActividad})} \quad \& \quad \text{FrecuenciaCardiacaPromedio}$$



OJO! con los tipos en composición

> TieneTaquicardia . hacerActividad

true

> hacerActividad , TieneTaquicardia

Rompe *

porque hacerActividad $\xrightarrow{\text{Int}}$ TieneTaquicardia ✓

pero TieneTaquicardia $\xrightarrow{\text{Bool}}$ hacerActividad X

hacerActividad no recibe un bool,

no todos los valores de bool son incluidos en int??

TieneTaquicardia Despues DE Entrenar unaFrecuencia = TieneTaquicardia , hacerActividad & unaFrecuencia

" "

:: Int \rightarrow Bool

P R E C E D E N C I A S	()
	Aplicación Prefija \longrightarrow TieneTaquicardia . <u>hacerAct 100</u> NP prefija

x/
+ -
==, /=
&&
||
\$ → lo ultimo que se resuelve

TIPO

Tipo : conjunto de valores asociado a un conjunto de operaciones

SUMA :: Int \rightarrow Int \rightarrow Int NO \longrightarrow SUMA 7,5 3,5 X Rompe

SUMA :: Float \rightarrow Float \rightarrow Float NO \longrightarrow SUMA 3,5 Frec.Card X rompe

Id :: $a \rightarrow a$ $\xrightarrow{\text{varable de tipo}}$ Devuelve lo mismo que recibe
val = valor

Const :: unTipo \rightarrow otroTipo \rightarrow unTipo $\xrightarrow{\text{recibe 2 cosas}}$
const valor1 valor2 = valor1 $\xrightarrow{\text{devuelve lo 1st}}$

SUMA :: Num a \Rightarrow a \rightarrow a \rightarrow a ✓ OK
Suma nro1 nro2 = nro1 + nro2

FromIntegral :: (Integral a, Num b) \Rightarrow a \rightarrow b \longrightarrow Es como castear de un Int a otro tipo + genérico

TíPICO DE LA COMPOSICIÓN

(.) una función otra función \circ valor = una función (otra función \circ valor)

11/4

Tuplas

- TAMANIO Fijo ("Aemera XL Negra", 1000) → duplo
 - DIFERENTES TIPOS DE DATOS ("Fiar", "Chronos", 2500000) → Terna
 - (no homogeneos)

Funciones para tuplas

FST } De vuelven 1º y 2º param en Duplas
Snd }

$$\begin{aligned} \text{fst} &:: (a,b) \rightarrow a \\ \text{snd} &:: (a,b) \rightarrow b \end{aligned}$$

Aplicar Descuento :: Num a \Rightarrow (String, a) \rightarrow a \rightarrow (String, a)

Aplicar Descuento : unProducto descuento = (fst unProducto , snd unProducto - descuento)

Aplicar descuento (nombre, precio) descuento = (nombre, precio - descuento)

que pasa si no uso uno de los componentes de tuplas?

PUEDO USAR —

producciónDeLugo :: Num a \Rightarrow (String, a) \rightarrow Bool

producto de lujos (nombre, λ) = elem "x" nombre || elem "z" nombre
VER. ANÓNIMA

0:0 NO poner - despues de = , solo va antes (Solo en mundo de patrones)

nombreFuncion :: TIPOS

nombre función patrones = valores

LISITAS

- no tiene TAMAÑO CONCRETO
- es homogénea (todo del mismo TIPO)

funciones para LISTAS

length Longitud de LISTA

reverse DARLA VUELTA

elem saber si hay un elemento presente \rightarrow elem :: Eq a \Rightarrow a \rightarrow [a] \rightarrow Bool
Tiene que ser Eq porque va comparando

head SACAR primer elemento

tail Devuelve LISTA sin primer elemento

MAX Devuelve maximo \rightarrow max :: Ord a \Rightarrow a \rightarrow a \rightarrow Bool
entre 2 cosas
Del mismo tipo

APLICACIÓN PARCIAL

max 'a' 'c' :: char \rightarrow aplicación TOTAL

max 'a' :: char \rightarrow char \rightarrow APLICACIÓN PARCIAL
función

Double Del Maximo :: Int \rightarrow Int \rightarrow Int

Double Del Maximo unNumero otroNumero = ((2 *), MAX, unNumero) otroNumero

ORDEN SUPERIOR

Recibe como Argumento otra función

- map
- filter
- \$
- • (composición)

18/4

métodos

ctrl + d → SIGUIENTE INSTANCIA

Foldable t \Rightarrow t a, \rightarrow Int
 lo veremos como [a] \rightarrow Int

Alt + Shift + Flecha \rightarrow SIGUIENTE LINEA
 : i INT me dice las FAMILIAS
 A LAS que pertenece INT

ALIAS DE TIPO

Si tengo que escribir muchos libros de tipo (String, String, Int)
 puedo crear un alias de tipo para evitar repetición

type Libro = (String, String, Int)

de hecho si igualo algo de tipo alias a su tipo devuelve true, porque en definitiva es lo mismo.

> Libro1 => Libro2
 true

Libro1 :: Libro
 Libro2 :: (String, String, Int)

Ahora si resolvemos el ejercicio :

type Biblioteca = [Libro]

prom Pags :: Biblioteca \rightarrow Int

prom Pags una Biblioteca = div(Sumatoria Pags Biblioteca una Biblioteca) / (length una Biblioteca)

porque la división (/) :: factorial n \Rightarrow n \rightarrow a \rightarrow a
 entonces no pueden ser los operandos INT
 y el resultado FLOAT

Sumatoria Pags Biblioteca una Biblioteca "div" length una Biblioteca

Las funciones normalmente
 prefijas pueden pasarse a infixas con //

Sumatoria Pags Biblioteca

Completo

lectura Obligatoria :: Libro → Bool

lectura obligatoria unLibro = esDeStephenKing unLibro ||
 esDeSagaEragon unLibro ||
 unLibro == fundacion

esDeStephenKing :: Libro → Bool

esDeStephenKing unLibro = autor unLibro == "Stephen King"

autor :: Libro → String

autor (unAutor, _, _) = unAutor

esDeSagaEragon :: Libro → Bool

esDeSagaEragon unLibro = elem (titulo unLibro) ["Eragon", "Eldest", "Legends", "Brisinger"]

titulo :: Libro → String

titulo (_, unTitulo, _) = unTitulo

ACCESOS

biblioteca Fantasiosa :: Biblioteca → Bool

bibliotecaFantasiosa unaBiblioteca = any esLibroFantasioso unaBiblioteca

esLibroFantasioso :: Libro → Bool

esLibroFantasioso unLibro = esdeAutor "Christopher Paolini" unLibro ||
 esdeAutor "Neil Gaiman" unLibro

esdeAutor :: String → Libro → Bool

esdeAutor unAutor unLibro = autor unLibro == unAutor

nombreDeLaBiblioteca :: Biblioteca → String

nombreDeLaBiblioteca unaBiblioteca = sinVocales . concat . listaTitulos \$ unaBiblioteca

completar

GUARDAS

genero1 :: Libro → String

genero unLibro

- | esDe Autor "stephenking" unLibro = "Terror"
- | esDe Autor Japones unLibro = "Manga"
- | Tiene Menos de 40 Paginas unLibro = "Comic"

OJO

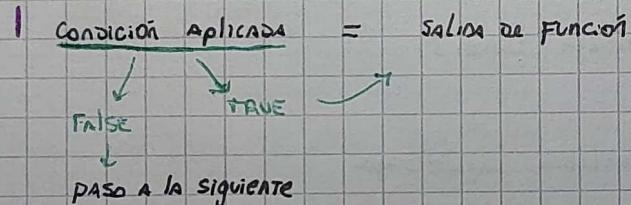
no se puede tener
diferentes tipos de
retorno

EVALUA EN
ORDEN

Si no cumple
PASA al SIGTE

Se podría poner otherwise pero en
este caso no hace falta

en CADA GUARDA :



Si bien SE PUEDE,
No DEVOLVER BOOLEANOS
EN UNA GUARDA
Error grave

- LAS GUARDAS SON MUTUAMENTE EXCLUYENTES
- Si encuentra el True, ya no sigue "buscando"

PATTERN MATCHING

[VER SI ENCAJA CON PATRON]

Veremos el ejemplo de LECTURA OBLIGATORIA , haciendolo por P.M.

Lectura Obligatoria :: Libro → Bool

Lectura Obligatoria unLibro = esDeStephenKing unLibro || esSagaEragon unLibro || unLibro == Fundacion

Type Saga = [Libro]

SagaDeEragon :: Saga

SagaDeEragon = [eldest, eragon, brisingr, Legado]

esSagaDeEragon :: Libro → Bool

esSagaDeEragon unLibro = elem unLibro sagaDeEragon

En el PATTERN MATCHING una vez que encaja con un PATRON, YA NO SIGUE BUSCANDO, LA SALIDA ESTARA DETERMINADA

Deberemos ir de lo + ESPECIFICO a lo gen.

Con PATTERN Matching

lecturaObligatoria	:: Libro → Bool	
Lectura Obligatoria	("StephenKing", -, -)	= True
Lectura Obligatoria	fundacion	= True
Lectura Obligatoria	unLibro	= esDeSagaEragon unLibro

ACA el orden importa porque si lo invierto entra todo por unLibro y nunca llega a evaluar fundacion

Si viene un fundacion en ese caso entraria por unLibro y devolvera false, nunca llegaría al patron de fundacion

Si tuviese mas de una saga pongo : Lectura Obligatoria unLibro = esDeSaga1 unLibro || esDeSaga2 unLibro ...

NOTA

Sin embargo ESTO NO FUNCIONA

no funciona porque fundación lo toma como PATRÓN, no como VALOR (eso iba a la derecha del =) entonces siempre entra por ahí y el resto de las opciones no se evalúan

DATA

```
type Persona = (String, String, Int)
```

```
juli :: Persona
```

```
juli = ("Julian", "Berbel Alt", 27)
```

```
gus :: Persona
```

```
gus = ("Gustavo", "Trucco", 30)
```

CUANDO DEFINO UN ALIAS DE TIPO NO ESTOY
CREANDO UN NUEVO TIPO, SINO SIMPLEMENTE
DANDOLE UN ALIAS A UN TIPO YA EXISTENTE



O SEA QUE ME DEJARÍA POR EJEMPLO
APLICARLE FUNCIONES DE ALIAS Libro
A LOS ALIAS Persona

- LOS TIPOS DE DATO SE CREAN CON DATA

```
data Persona = Una Persona String String Int  
CONSTRUCTOR
```

```
juli :: Persona
```

```
juli = UnaPersona "Julian" "Berbel Alt" 27
```

```
gus :: Persona
```

```
gus = UnaPersona "Gustavo" "Trucco" 30
```

Entonces si pongo ahora Libro como data:

```
data Libro = UnLibro {  
    autor :: String  
    titulo :: String  
    paginas :: Int  
}  
deriving (Eq, Show)
```

Definiendo así!
EVITO USAR LOS ACCESORES!

PARA QUE PUEDAN
SER COMPARABLES
Y MOSTRABLES

⇒ Los data A DIFERENCIA DE LAS TUPLAS SE PUEDE PONER TIPOS DE LISTA (ej. [Libro])

Si quiero hacer una función que "modifique" un dato (entre comillas porque en realidad no se modifica sino que se crea una copia)

sacarSecuela :: Libro → Libro

SacarSecuela unLibro = unLibro { TITULO = titulo unLibro ++ "2",
paginas = paginas unLibro + 50 }
"Modifico" solo
2 atributos del dato

En resumen cuando creo un dato estoy creando

Tipo → "Persona"
Patrón → "Una Persona"
Valor → "Una Persona"

Hasta ahora vimos 3 formas de crear funciones:

Típica $\text{siguiente } \text{unNum} = \text{unNum} + 1$

Composición doble. siguiente

Aplicación parcial $(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
 $(+) 2 :: \text{Int} \rightarrow \text{Int}$

data

y veremos una 5^{ta}:

EXPRESIONES LAMBDA

- CREA una función "Al Vuelo" sin darle un nombre → no hace falta tiparla

$\text{siguienteDeLista} :: [\text{Int}] \rightarrow [\text{Int}]$ LAMBDA

$\text{siguienteDeLista } \text{unosNumeros} = \text{map } (\lambda \text{unNumero} \rightarrow \text{unNumero} + 1) \text{ unosNumeros}$

precondiciones para usar Lambda:

- Solo la usamos una vez
- No tenemos un buen nombre para la lógica. (funciones muy abstractas)

CURRIFFICACIÓN (por el inventor Haskell Curry)

Sumade3 :: Int → Int → Int → Int
 $\lambda n_1 \ n_2 \ n_3 = n_1 + n_2 + n_3$

TODOS lo mismo

Sumade3' :: (Int → Int → Int → Int)
 $\lambda n_1 \ n_2 \ n_3 \Rightarrow n_1 + n_2 + n_3$

Sumade3'' :: (Int → (Int → (Int → Int)))
 $\lambda n_1 \rightarrow \lambda n_2 \rightarrow \lambda n_3 \rightarrow n_1 + n_2 + n_3$

Sumade3''' :: (Int → (Int → (Int → (Int → Int))))
 $\lambda n_1 \rightarrow \lambda n_2 \rightarrow \lambda n_3 \rightarrow n_1 + n_2 + n_3$

Haskell cuando definimos una función de N parámetros

Se convierte en una cadena de N funciones de 1 parámetro

es decir que para Haskell las funciones de >1 parámetros no existen

La consecuencia de la curificación es la Aplicación parcial

y por la curificación es que las funciones que componemos son de 1 parámetro

(.) :: (b → c) → (a → b) → (a → c)

(.) f g = \x → f(gx)

↓ ADDRÍA SET

(.) :: (b → c) → (a → b) → a → c

(.) f g x = f(gx)

RECUSIVIDAD

ES UNA FUNCIÓN QUE SE LLAMA A SI MISMA. TIENE

- CASO RECURSIVO: lo que se repite
- CASO BASE: una forma de salir

FACTORIAL :: Int → Int

FACTORIAL 0 = 1 → CASO base

FACTORIAL numero = numero * FACTORIAL (numero - 1) → CASO RECURSIVO

fibonacci

fibonacci 0 = 0

fibonacci 1 = 1

Fibonacci

complejo

PATRONES PARA LISTAS

[null] :: [a] → Bool

null [] = True. > patrón de lista

null _ = False

[] → LISTA VACÍA

(x : xs) → Primer y último elemento

(x₁ : x₂ : x₃ : xs) → mínimo 3 elementos

[head] :: [a] → a

head (cabeza : _) = cabeza

head [] = undefined

[tail] :: [a] → [a]

tail (_ : cola) = cola

tail [] = undefined

REPASO DE FUNCIONES

(:) → Agrega cabecera a la lista

(++) → Concatena listas

1 : [2,3] → [1,2,3]

1 : (2 : []) → [1,2]

FoLD

redefinimos algunas funciones conocidas de manera recursiva:

Sum :: Num a \Rightarrow [a] \rightarrow a

Sum [] = 0

Sum (cabeza : cola) = cabeza + sum cola

All :: (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool

All [] = True

All (predicado (cabeza : cola)) = predicado cabeza & all predicado cola

Def All (>3) [1,2,3,4]
=> False

length :: [a] \rightarrow Int

length [] = 0

length (_ : cola) = 1 + length cola

map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]

map [] = []

map transformacion (cabeza : cola) = transformacion cabeza : map transformacion cola

puedo reescribir el caso recursivo de todos

	operacion entre cabeza y cola	Algo sobre la cabeza	Algo sobre la cola
sum	$\rightarrow (+)$	cabeza	(sum cola)
all	$\rightarrow (\& \&)$	predicado cabeza	all predicado cola
length	$\rightarrow (+)$	1	length cola
map	$\rightarrow (:)$	transf cabeza	map transf cola

Vemos que la ESTRUCTURA es común a todos, por lo que haremos una función que abstracta esta lógica.

funcion :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b

funcion valorBase [] = valorBase

funcion operacion valorBase (cabeza : cola) = (operacion) (cabeza) (funcion operacion valorBase cola)

Esta función ya está en Haskell y se llama foldr

Sum :: Num a \Rightarrow [a] \rightarrow a

Sum LISTA = foldr (+) 0 LISTA

NOTA

LISAS INFINITAS

Haskell me deja trabajar con LISTAS INFINITAS

[1...] → lista de numeros ∞

[2,4...] → lista de numeros ∞ pares

duda: rachar variable

inda: semilla

duda: componek nappus

duda: fundacion

head [1...] → 1

last [1...] → se queda pensando

tail [1...] → LISTA INFINITA SIN EL 1

Este modo de comportarse tiene que ver con la forma en la que evalua Haskell

LAZY EVALUATION

Lazy Evaluation



EAGER EVALUATION

- "Evaluación perezosa"

- El que usa Haskell

- "Evaluación ansiosa"

- El que usan casi todo el resto de los lenguajes

- Estrategia call by NAME

↓
primer analiza que tiene
que resolver

ej: haciendo el head de lista infinita
no le hace falta terminar de evaluar la lista
porque necesita solo el 1º

- Estrategia call by VALUE

↓
antes de resolver tiene que
"saber sus parámetros"

ej: haciendo el head de lista infinita primero
se evalua la lista y luego el head (nunca podrá la lista)
∴ queda colgada

Lazy Evaluation

`take 15 [1..]`

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

no puede, queda colgada

`last 15 [1..]`

quedó colgada

`length [1..]`

quedó colgada

`Sum [3, 6.., 3*10]`

165

165

`any Even [2, 4..]`
algún par...

true

quedó colgada

`all even [2, 4..]`

quedó colgada

`all odd [2, 4..]`

quedó colgada

`filter (<3) [1..]`

"[1, 2]" y nunca termina
se queda pensando

quedó colgada

Employa a responder pero
nunca termina (LISTA ∞)

`map (*2) [1..]`

quedó colgada

"Hola"

`fst ("Hola", [1..])`

tuple

quedó colgada

"Hola"

`fst ("Hola", 7/0)`

quedó colgada

"Hola"

`Snd ([1, "Hola"], 2)`

Rompe, no se pide ÷ por 0

Rompe, porque ya es un
error de tipos, no se
puede más de un tipo en la
LISTA, NI siquiera compila

Eager Evaluation

Fold

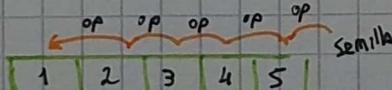
Fold R

$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
 $\text{foldr} = \text{Semilla} [] = \text{Semilla}$
 $\text{foldr} \quad \text{op} \quad \text{Semilla} (x:xs) =$
 $\text{op} \times (\text{foldr} \text{ op} \text{ Semilla} \ xs)$

$\text{foldr} (+) 0 [1,2,3,4,5]$

$(+) 1 (\text{foldr} (+) 0 [2,3,4,5])$
 $(+) 1 ((+) 2 (\text{foldr} (+) 0 [3,4,5]))$
 $(+) 1 ((+) 2 ((+) 3 (\text{foldr} (+) 0 [4,5]))))$
 $(+) 1 ((+) 2 ((+) 3 ((+) 4 (\text{foldr} (+) 0 [5]))))$
 $(+) 1 ((+) 2 ((+) 3 ((+) 4 ((+) 5 (\text{foldr} (+) 0 [])))))$
 $(+) 1 ((+) 2 ((+) 3 ((+) 4 ((+) 5 0))))$
 $(+) 1 ((+) 2 ((+) 3 ((+) 4 5))))$
 $(+) 1 ((+) 2 ((+) 3 9))))$
 $(+) 1 ((+) 2 12))$
 $(+) 1 14)$

15



Fold L

$\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
 $\text{foldl} = \text{Semilla} [] = \text{Semilla}$
 $\text{foldl} \quad \text{op} \quad \text{Semilla} (x:xs) =$
 $\text{foldl} \quad \text{op} \quad (\text{op} \text{ semilla} \ x) \ xs$

$\text{foldl} (+) 0 [1,2,3,4,5]$

$(\text{foldl} (+) (+) 0 1) [2,3,4,5]$
 $(\text{foldl} (+) (+) (+) 0 1) 2) [3,4,5]$
 $(\text{foldl} (+) (+) (+) (+) 0 1) 2) 3) [4,5]$
 $(\text{foldl} (+) (+) (+) (+) (+) 0 1) 2) 3) 4) [5]$
 $(+) ((+) ((+) ((+) 0 1) 2) 3) 4) 5)$
 $(+) ((+) ((+) ((+) 1 2) 3) 4) 5)$
 $(+) ((+) ((+) 3 3) 4) 5)$
 $(+) ((+) 6 4) 5)$
 $(+) 10 5)$

15



Si no puedo incluir una semilla uso:

FoldR1

$\text{foldR1} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$

$\text{foldR1 op } (x:xs) = \text{foldR op } x \ xs$

FoldL1

$\text{foldL1} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$

$\text{foldL1 op } (x:xs) = \text{foldL op } x \ xs$

FUNCIONES de GENERACIÓN de LISTAS

[repeat] :: $a \rightarrow [a]$ me crea una lista ∞ del elemento

[iterate] :: $(a \rightarrow a) \rightarrow a \rightarrow [a]$ va aplicando la función a ese elemento

iterate (*) 2 [2, 4, 6, 8... y así. Lista ∞ (podría no ser ∞ si le aplico por ej un take 5)]

[Replicate] :: $\text{Int} \rightarrow a \rightarrow [a]$

Dado un numero de reps y un elemento me crea una Lista con ese numero de reps

[cycle] :: $[a] \rightarrow [a]$

Crea una lista ∞ repitiendo la lista dada

MAPPERS

Dado un dato puedo hacer mappers para evitar repetir lógica

```
data Persona = Persona {
    nombre :: String
    apellido :: String
    edad :: Int
}
```

deriving Show

Si tengo por ejemplo:

cumplirAños unaPersona = unaPersona {edad} = edad unaPersona + 1

duplicarEdad unaPersona = unaPersona {edad} = edad unaPersona * 2 }

copiarPersona unaPersona otraPersona = unaPersona {edad} = edad otraPersona,
apellido = apellido otraPersona }

Lo puedo facilitar en:

mappers para dato

mapedad :: $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Persona} \rightarrow \text{Persona}$

mapedad unaFuncion unaPersona = unaPersona {edad} = unaFuncion.edad unaPersona }

quedando

cumplirAños unaPersona = mapEdad (+1) unaPersona

NOTA

para componer mappers :

copiar Persona :: Persona → Persona → Persona

copiar Persona unaPersona otraPersona = mapEdad (const (edad otraPersona)),
mapApellido (const (apellido otraPersona)) \$ unaPersona

uso const porque si osi necesito una función para
el mapper (const toma dos valores y
retorna el 1º)

Otro ejemplo:

cumplir 100 Años unaPersona = mapEdad (const 100) unaPersona

La ventaja de los mappers es que si quiero cambiar el modelo de datos por ejemplo
un dato por tuplas **solo cambia los mappers**

Mapper para tuplas

mapEdad :: (Int → Int) → Persona → Persona

mapEdad unaFunción (nombre, apellido, edad) = (nombre, apellido, unaFunción edad)

PARADIGMA FUNCIONAL (HASKELL)

- Funcional viene de función, que tiene dominio e imagen y cumple con unicidad y existencia.

Haskell (Archivos .hs)

- Valores

NÚMEROS 123 → NÚMEROS NEGATIVOS
 CON (-) EN CONSTRUCCIÓN
 BOOLEANOS TRUE, FALSE
 STRINGS "hola"
 CARACTERES 'a'

Funcional → pureza
(NADA SE MODIFICA)

Ej: SIN Funcional CREADOS NUEVOS MUNDOS
↓ AUTO CON NAFTA

Con funcional → NUEVO AUTO
CREADO
+ AUTO ANTERIOR

- funciones +, -, ||, &&, etc. ya vienen con haskell

- DEFINIR Función Propia

SIGUIENTE → ARGUMENTO; que le pasa como parámetro
 NOMBRE → UN NUMERO → UN NUMERO + 1 → lo que hace la función
 = AGREGAR
 FUNCIÓN

y luego debes poner en terminal :l (de reload) y luego ya puedo poner por ejemplo "SIGUIENTE 1" y me mostrará "2"

- CREAR NUEVO PROYECTO

STACK NEW NOMBRE DE PROYECTO (sin espacios) → CREAR PROYECTO

cd NOMBRE DE PROYECTO → PARA IR A CARPETA DE PROYECTO

STACK BUILD ghci → para que se "ACOMODE todo"
 ↓
 STACK GHCI LIB.hs → INTERPRETER

GHCI → Compilador de Haskell (como el año pasado teníamos GCC para C++)

TERMINAL → INTERFAZ + BÁSICA POR LA QUE PUEDES INTERACTUAR CON MI SISTEMA OPERANDO

expresión: Algo que se puede evaluar y da un resultado. Una función es una expresión porque tiene existencia.

- Siguiente UN NÚMERO → NOTACIÓN PREFIJA
- UN NÚMERO + 1 → NOTACIÓN INFIXA
- Aquí en Haskell puedes tener igual nombre de funciones, ya que el ámbito es como en C++
Variables en

TIPADO

Haskell trabaja con Inferencia de tipos, deduce el tipo de dato que uso pero igual podemos decirle que tipo es:

Variables (ctes)	e :: (Float) e = 2,718281	mNombre :: String mNombre = "Lucas"	Float Bool Int String
Fuciones	Siguiente :: Int → Int Siguiente UN NÚMERO = UN NÚMERO + 1	Recebe → devuelve Recebe → devuelve	MULTIPLICAR :: Int → Int → Int MULTIPLICAR UN NÚMERO OTRONUMERO = UN NÚMERO * OTRO NÚMERO

En Funcional NO IMPORTA el orden de las líneas

No puede cambiar el valor de los datos en tiempo de ejecución
(NO hay variables)

- Si pongo $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{float}$ DA ERROR en la suma x ejemplo $\text{float} \rightarrow \text{float} \rightarrow \text{float} \rightarrow \text{Int}$ pq es cerrada.
para que no de error no debe ser una op. cerrada.
- Si pongo en la terminal :t Nombredefunción Me dice el tipo que le asigné
- Si no pongo el tipo de dato de la función me devuelve

$$\text{Num } a \Rightarrow a \rightarrow a \rightarrow a \rightarrow a$$

NOTA

ej: Suma de tres :: Num a \Rightarrow a \rightarrow a \rightarrow a \rightarrow a MISMO TIPO

Suma de tres UN NUMERO UNNUMERO2 UNNUMERO3 = UNNUMERO + UNNUMERO2 + UNNUMERO3

función const: De dos valores devuelve el primero de ellos (ej: terminal: const 5 6)
RETORNA 5

EN MI PROGRAMA SE PODRÁ PONER COMO:

CONSTANTE :: UNTIPOCUALQUIERA \rightarrow OTROTIPOCUALQUIERA \rightarrow UNTIPOCUALQUIERA
 CONSTANTE UNValor OTROVALOR DOS TIPOS
DIFERENTES

Restringiendo la suma a solo números

SUMA :: NUM a \Rightarrow a \rightarrow a \rightarrow a
 SUMA UN NUMERO OTRO NUMERO = UN NUMERO + OTRONUMERO

NUM en este caso es una "FAMILIA". TODOS LOS DE ESA FAMILIA (NUM, INT, FLOAT, Double, Integer) PUEDEN SUMARSE, RESTARSE Y MULTIPLICARSE (+) (-) (*)

Otro ejemplo
 $= =$ es tipo $(==)$:: Eq a \Rightarrow a \rightarrow a \rightarrow Bool

Eq es otra "FAMILIA": Int, Float, Bool, String, Char, ...
 Y A TODOS SE LE PUEDE HACER igual a, DIFERENTE a
 $(==)$ (\neq)

COMO NADA PUEDE CAMBIAR, EN FUNCIONAL TODO SE PASA POR VALOR,
 NADA POR REFERENCIA

ORD es otra "FAMILIA": Int, Float, Char, String
 Y A TODOS SE LE PUEDE HACER MAYOR a, MENOR a
 $(>)$ $(<)$

$(>)$:: Ord a \Rightarrow a \rightarrow a \rightarrow a \rightarrow Bool

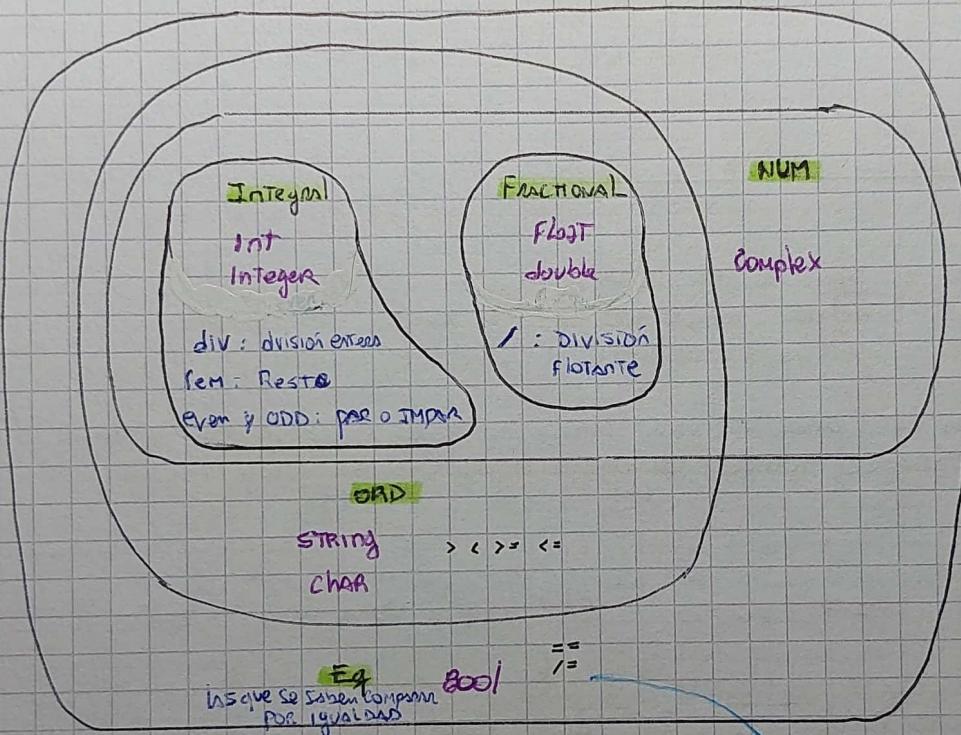
• + Familia
 • ! para ver las ops que involucra esa familia

PARA HACER MUCHAS RESTRICCIONES DE TIPO

es Mayor De Edad :: $\{(Ord\ a, Num\ a)\} \Rightarrow a \rightarrow \text{Bool}$
es Mayor De Edad UNA EDAD = UNA EDAD ≥ 18

Debo Restringir P. También Ord para comparar con Mayor o Igual
porque no todos los Números son ordenables (ej: complejos)

Familias de Tipos



Hay Eq sin show y
NAY show que no son
Eq. (pero no
se ven)

Show: Los que se pueden
mostrar por pantalla

$a \rightarrow b$
Las funciones
(o sea que las funciones son
valores)

- para Mostrar funciones en pantalla pongo al principio del código

```
IMPORT Text.Show.Functions
```

porque sino no están configuradas para mostrarse

NOTA

21/4

⑥ SUMATRAS $a \ b \ c = a + b + c$

→ ¿Como se que tipos de dato son a, b, c ? porque $+$ es tipo Num, entonces no puede ser una letra.

• Si fuera división por ejemplo, se que es un fractional.

⑦ SUMATRAS $a \ b \ c = \underbrace{a + b + c}_{\text{num}} + \underbrace{\text{longitud "holi"} \downarrow}_{\text{int}}$ (habiendo definido longitud)

COMPOSICIÓN DE FUNCIONES

Siguiente :: Int → Int

Siguiente UN NÚMERO = UN NÚMERO + 1

MAS DOS :: Int → Int

MAS DOS UN NÚMERO = Siguiente (Siguiente UN NÚMERO)

} + complicado

Haskell tiene UNA función para composición:

$f \circ g$ en Haskell. $f \circ g$

Siendo $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

entonces yo puedo hacer:

$$\begin{aligned} \text{MAS DOS UN NÚMERO} &= \underbrace{\text{Siguiente}}_f (\underbrace{\text{Siguiente}}_g \underbrace{\text{UN NÚMERO}}_x) \\ &\quad \downarrow \quad \downarrow \quad \downarrow \end{aligned}$$

$$\text{MAS DOS UN NÚMERO} = (\text{Siguiente} . \text{Siguiente}) \text{ UN NÚMERO}$$

$$f(gx) = (f \circ g)x$$

NOTA

- Si quiero hacer más tres:

Como componer
3 funciones.
o my

MASTRES :: INT → INT

MASTRES UN NUMERO = (siguiente, siguiente, siguiente) UN NUMERO

MASTRES UN NUMERO = (siguiente, MASTRES) UN NUMERO.

- OJO con los tipos, el tipo de a = tipo de a y el tipo de c = tipo de c.

ej: Length . Even ~~NO~~ se puede, la imagen de even no entra en el dominio de length. No puedo entrar a el length con un true o false.

- Teniendo SumaTres :: INT → INT → INT → INT
Suma de Tres a b c = a + b + c

Si pongo.

SumaTres 6 Devuelve <function> porque no está completa.

Si ponemos

: t SumaTres 6 Devuelve SumaTres 6 :: INT → INT → INT

Me va complementar los int

Cuando a una función le paso menos parámetros de los que espera, se dice que la estoy APLICANDO PARCIALMENTE.

- Si le pongo : t CONST TRUE
Devuelve CONST TRUE :: b → Bool

Porque una vez que le di un booleano, lo que devuelve tiene que ser obligatoriamente un booleano.

PRECAUCIÓN: CONST es una función que recibe dos parámetros y devuelve el valor del primero.

- Puedo definir también "Siguiente" como aplicación parcial.

Siguiente :: Int → Int

Siguiente UN NÚMERO = $(\lambda x. 1)$ UN NÚMERO

$\equiv 1 + \text{UN NÚMERO}$

Le estoy pasando el 1 como primer parámetro de la suma, y completa el segundo parámetro que le falta con "Un Número".

Se ve así:

$$\begin{array}{ll} (+) & :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a \\ (+) 1 & :: \text{Num } a \Rightarrow a \rightarrow a \\ (+) 15 & :: \text{Num } a \Rightarrow a \end{array}$$

$$\begin{array}{l} (+) 2 = 2 + \\ + 2 \stackrel{\text{equiv}}{=} + 2 \end{array}$$

- Si quiero hacerlo con $-$, en vez de $(- 5)$ hago substract 5 Llamada como -5

Simplificación

Siguiente :: Int → Int
 ↓ tipo de retorno
 Siguiente UN NÚMERO = $(\lambda x. 1)$ UN NÚMERO
 ↓ ↓ ↓
 UNA FUNCIÓN PARÁMETRO OTRA FUNCIÓN EL MISMO PARÁMETRO

entonces esta segunda línea también la puedo escribir como

Siguiente :: Int ^{tipo de retorno}
 Siguiente = $(\lambda x. 1)$
 ↓
 solo el tipo del retorno

Otro ejemplo

esMultiplo :: Int → Int → Bool

esMultiplo múltiplo UN NÚMERO = $(\text{EsIgualACero}, \text{mod Múltiplo})$ UN NÚMERO

esMultiplo múltiplo = EsIgualACero . mod Múltiplo

Ojo: Aquí no hay aplicación parcial.
 La aplicación es solo del lado derecho.

NOTA de 11:30 - 11:54

A LAS FUNCIONES QUE RECIBEN OTRAS FUNCIONES COMO PARÁMETRO (COMO .) SE LAS LLAMA FUNCIONES DE ORDEN SUPERIOR

OTRAS FUNCIONES DE ORDEN SUPERIOR

\$

$$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$$

PRECEDENCIAS

DE OPERADORES

- PARENTESIS
- APLICACIÓN PREFIXA
- \circ
- $++ - * / \& \& \parallel \parallel$
- $\$$

\$ Sirve para modificar precedencias

CASO 1 (doble.length) "holi": doble.length y a eso se le aplica a "holi"

PARENTESIS

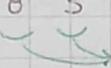
CASO 2 doble. length "holi": length "holi" y eso se compone con doble
AP. PREFIXA

CASO 3 doble. length \$ "holi": MISMO que CASO 1, me ahorro el PARENTESIS

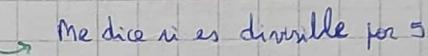
flip

$$\text{flip} :: (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$$

EVALUA LA FUNCION CON LOS PARAMETROS DADOS YUELVA.

Si hago $\text{flip } (-) 8 3$
 se vuelve -5  no hace falta que sean del mismo tipo

Es un cuando quiero COMPOSER funciones Aplicando parcialmente el segundo parametro

Ejemplo $\text{mod } 10 5$
 $\Rightarrow 0$
 $(\text{flip mod } 5) 10$ 
 $\Rightarrow 0$

Una aplicación puede ser ... en vez de esto:

$\text{SALUDAR} :: \text{String} \rightarrow \text{String}$
 $\text{SALUDAR UN NOMBRE} = "Hola" ++ \text{UN NOMBRE} ++ "!"$

$\text{SALUDAR FORMALMENTE} :: \text{String} \rightarrow \text{String}$
 $\text{SALUDAR FORMALMENTE UN NOMBRE} = "Hola Señor" ++ \text{UN NOMBRE} ++ "!"$

$\text{SALUDAR EGRESADO UTN} :: \text{String} \rightarrow \text{String}$
 $\text{SALUDAR EGRESADO UTN UN NOMBRE} = "Hola Ing." ++ \text{UN NOMBRE} ++ "!"$

• Define ESTO:

Función auxiliar de orden superior
A

$\text{SALUDAR SEGUN} :: (\text{String} \rightarrow \text{String}) \rightarrow \text{String} \rightarrow \text{String}$

SALUDAR SEGUN UN NOMBRE = "Hola" ++ UNA FUNCION UN NOMBRE ++ "!"

y Luego se me facilita hacer las otras 3

$\text{SALUDAR} :: \text{String} \rightarrow \text{String}$
 $\text{SALUDAR UN NOMBRE} = \text{SALUDAR SEGUN id UN NOMBRE}$

$\text{SALUDAR FORMALMENTE} :: \text{String} \rightarrow \text{String}$
 $\text{SALUDAR FORMALMENTE UN NOMBRE} = \text{SALUDAR SEGUN} ("Señor" ++ UN NOMBRE)$

(lo mismo con SALUDAR EGRESADO UTN)

28/4

OTRA FORMA De Definir funciones :

Escribir función de forma anónima

Siguiente 3

$\Rightarrow 4$

Siguiente número \rightarrow número + 1

$(\lambda \text{ NÚMERO} \rightarrow \text{NÚMERO} + 1) \ 3$

$\Rightarrow 4$

Expresión Lambda: Es una función que no tiene nombre.

$(\lambda \text{ NÚMERO1 NÚMERO2} \rightarrow \text{NÚMERO1} + \text{NÚMERO2}) \ 600$ \rightarrow equivale a $600 + \text{un número}$.
(AP PARCIAL)
 $\text{Número 2} = 600 + \text{Número 2}$

$(\lambda \text{ NÚMERO1} \rightarrow (\lambda \text{ NÚMERO2} \rightarrow \text{NÚMERO1} + \text{NÚMERO2}))$



Así es como lo lee Haskell

Esto es la CURRIFICACIÓN, Toda función en Realias Recibe UN parámetro y Devuelve otra función que Recibe otros parámetros y así... (para entenderlo, Haskell y el paradigma funcional funcionan de a un parámetro a la vez).

Puedo aplicar la aplicación parcial gracias a que existe la currificación

suma :: Num a $\Rightarrow (\lambda a \rightarrow (\lambda a \rightarrow a))$

$\lambda \ominus$

suma :: Num a $\Rightarrow a \rightarrow a \rightarrow a$

Composición $f g x = f(gx)$

Composición $f g = \lambda x \rightarrow f(gx)$

Definir Función por partes (GUARDAS)

Si tengo que definir la función módulo:

$$\text{abs}(x) = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{si } x < 0 \end{cases}$$

haskell TAMBIÉN ME DEJA DEFINIR UNA FUNCIÓN POR PARTES:

$$\Rightarrow \text{ValorAbsoluto} :: (\text{Num}, \text{a}) \Rightarrow \text{a} \rightarrow \text{a}$$

\Rightarrow **VALORABSOLUTO** NUMERO

$$\begin{array}{l|l} \text{TAB} & \begin{array}{l} \text{NUMERO} \geq 0 = \text{NUMERO} \\ \text{NUMERO} < 0 = -\text{NUMERO} \end{array} \end{array}$$

$$\Rightarrow \text{Signo} :: (\text{Ord}, \text{Num}, \text{a}) \Rightarrow \text{a} \rightarrow \text{a}$$

\Rightarrow **SIGNO** NUMERO

$$\begin{array}{l|l} & \begin{array}{l} \text{NUMERO} < 0 = 1 \\ \text{NUMERO} \geq 0 = 1 \\ \text{NUMERO} == 0 = 0 \end{array} \end{array}$$

$$\Rightarrow \text{DISCRIMINANTE} (b^2 - 4ac) = \begin{cases} = 0 & \text{RAIZ Doble} \\ > 0 & \text{TODOS Raices Simples} \\ < 0 & \text{Raices Complejas} \end{cases}$$

$$\text{DISCRIMINANTE } a \ b \ c = b^2 - 4 * a * c$$

$$\text{TIPO DE RAICES } a \ b \ c$$

$$\begin{array}{l|l} & \begin{array}{l} \text{DISCRIMINANTE } a \ b \ c == 0 = \text{"RAIZ Doble"} \\ \text{DISCRIMINANTE } a \ b \ c > 0 = \text{"TODOS Raices Simples"} \\ \text{DISCRIMINANTE } a \ b \ c < 0 = \text{"Raices Complejas"} \end{array} \end{array}$$

Importa el
ORDEN

Si entra por
el PRIMER CASO
no hace el 2^{do}

También hay un → "en cualquier otro caso" → **otherwise**. (se pone al final)

↓ Si no quiero definir ahí la función no pongo el caso y listo

PATTERN Matching

gusto de Helado Favorito a

a == "Feli"	= "chocolate"
a == "Fede"	= "VAINILLA"
a == "Giz"	= "ODL"
a == "Marco"	= "TRAMONTANA"
otherwise	= "NADA Rico"

para valores concretos (los que se pueden comparar con ==)

gusto de Helado Favorito' "Feli" = "chocolate"

gusto de Helado Favorito' "Fede" = "Vainilla"

gusto de Helado Favorito' a = "NADA Rico"

TAMBÍEN
Importa
el orden

↓
es como
un otherwise.

Si no usa "a"
a la derecha del
 igual como en
 este caso pienso

hacer: gusto de Helado Favorito _ = "NADA Rico"

Los tipos de datos que conocemos muchísimas veces no sirven para modelar lo que quiero.

PARA DATOS COMPLEJOS USO:

TUPLAS (Agrupar datos)

- SE PUEDEN HACER TUPLAS DE 3 pero Haskell tiene muchas funciones para **tuplas de 2 elementos**

("Juli", 26)

funciones **FST** → Devuelve 1º elemento
SND → Devuelve 2º elemento

SALUDAR :: (String, Int) → String

SALUDAR UNA PERSONA = "Hola" ++ **FST** UNA PERSONA

- Puedo nombrar una tupla

Juli :: (String, Int)

Juli = ("Julian", 26)

franco = ("Franco", 1733709)

→ **RECORDER** que en funcional las cosas no pueden cambiarse

Si lo quisiese cambiar no puedo, pero debo hacer otra función que me devuelva otra persona pero con la edad cambiada, es decir, similar pero no lo mismo

CUMPLIR AÑO :: (String → Int) → (String, Int)

CUMPLIR AÑO UNA PERSONA = (**FST** UNA PERSONA, **SND** UNA PERSONA + 1)

Si yo ahora pusiese "Juli" en consola, será el viejo, el de 26, no el nuevo de 27.

En funcional el mundo NO CAMBIAS

⇒ Muchas veces no terminan de servirme las tuplas porque por ejemplo Juli es UNA PERSONA CUALQUIERA Y Franco es UN ALUMNO.

NOTA → Tendría sentido sumarle 1 al legajo de Franco, si a la edad de Juli ESTUDIA BUENO HACER EXCLUSIVO A PERSONAS o EXCLUSIVO A ALUMNOS

Alias de tipo

type PERSONA = (Nombre : String, Edad : Int)

type ALUMNO = (String, Int)

e incluso
También

type Nombre = String
type Edad = Int

Este NO ES CREAR un tipo de dato, solo ponerle un alias a un tipo de dato por lo que no soluciona mi problema

EL problema se soluciona CREANDO un tipo de DATO NUEVO CON:

DATA

data Persona = Persona String Int deriving (Show)

me es lo mismo

para que sea mostrable x pantalla
(aumentar el
engo de show)

“La forma de construir una persona es poner “Persona”
(podría ser cualquier otra palabra) y luego un String e Int”

en consola → persona "Julian" 26

o en el código Juli = Persona "Julian" 26

- yo puedo preguntar que tipo es persona con :t → Persona :: String → Int → Persona

CREAR UN NUEVO TIPO DE DATO significa que
“persona” ya no es una tupla.

- Antes si quería mostrar uno de los datos usaba fst o snd, pero ahora?

nombre Persona :: Persona → String

nombre Persona (Person nombre edad) = nombre

formula de construcción

→ PATTERN MATCHING
CON DATA

edad Persona :: Persona → Int

edad Persona (Person nombre edad) = edad

PARA MOSTRAR
UNO DE LOS CAMPOS

nombre Alumno (nom, Leg) = nom

Legajo Alumno (nom, Leg) = Leg

→ PATTERN MATCHING
EN TUPLA
(haciendo definibles tipos)

Valores → A la derecha del igual

Patrones → A la izquierda del igual

Tipos → en donde hay ::

NOTA

Solo en Haskell se me facilita la escritura porque me deja hacer:

```
type Nombre = String
type Legajo = Int
data ALUMNO = ALUMNO {
    NombreALUMNO :: Nombre,
    LegajoALUMNO :: Legajo
} deriving (Show)
```

Me ahorro especificar los tipos luego.

```
data Persona = Persona {
    NombrePersona :: String,
    EdadPersona :: Int
} deriving (Show)
```

Luego la función Cumplir Año será:

CUMPLIRANIO :: Persona → Persona

opc 1 CUMPLIRANIO (Persona nombre edad) = Persona nombre (edad + 1)

opc 2 CUMPLIRANIO UNAPERSONA = PERSONA (nombrepersona unaPERSONA)(edadPERSONA unaPERSONA + 1)

USAR ENUM con DATA

data DiaDeSemana = Lunes | Martes | Miércoles | Jueves | Viernes | Sábado | Domingo deriving (Show)

esFinDeSemana :: DiaDeSemana → Bool

esFinDeSemana SABADO = True
esFinDeSemana Domingo = True

↓
Cree un tipo de dato con muchos valores

Así está definido el tipo de dato Bool

data Booleano = Falso | Verdadero deriving (Show)

Conjunción :: Booleano → Booleano → Booleano

Conjunción Verdadero Verdadero = Verdadero

LISTAS

Dentro de la lista tiene que ser todo del mismo tipo.

[1, 2, 3, 4, 5, 6] o [1..6]

[("Julian", 26), ("Fede", 33)]

[("AGA", 2014, [9, 10, 8]), ("AMI", 2015, [7, 8, 10])] (LISTAS dentro de LISTAS)

[] Lista vacía.

['h', 'o', 'l', 'a'] el STRING es UNA LISTA de caracteres

Length: Si la miro me da la cantidad de elementos

++: Concatenar listas

MAXIMUM: Me muestra el maximo

MINIMUM: Me muestra el minimo

Poniendo | Import Data.List | al principio tengo mas funciones como:

SUM: Suma todos los elementos

SORT: Ordena la lista

SORT ON: Ordena según un criterio.

Sorton length → me lo ordena por longitud

NULL: Me dice si la lista está vacía (true) → llena (false)

UNION: Me une dos conjuntos (mira los repetidos)

NUB: Me saca los elementos repetidos

elem + Atgo: Me dice si ese está o no en la lista

(!!) + Lista + numero: Me devuelve un elemento

DELETE + elemento + lista: Me saca ese elemento de la lista

take + n: Me toma los primeros n elementos

drop + n: Descarto los primeros n elementos

Concat: Lista de Listas : me da una lista con todos los elementos concatenados
reverse Lista : me devuelve una lista pero vuelta

HOJA N°

FECHA

head: nos da el primer elemento

Tail: nos da del segundo elemento hasta el final en una lista

Función Filter : Es del tipo filter :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow [a]$

Recibe una función y una lista y me devuelve una lista del mismo tipo

All + función + Lista : Me va a decir si todos son true de esa función o no.

Any + función + Lista : Me va a decir si Alguno elemento es true de esa función.

12/5

RECUSIVIDAD

FACTORIAL :: Int \rightarrow Int

OPC 1 (FACTORIAL 0 = 1) $\xrightarrow{\text{CASO BASE}}$
PATTERN MATCHING (FACTORIAL m = m * FACTORIAL (m-1)) $\xrightarrow{\text{CASO RECURSIVO}}$

OPC 2 (FACTORIAL m
Guardas | m = 0 = 1 $\xrightarrow{\text{CASO BASE}}$
| otherwise = m * FACTORIAL (m-1) $\xrightarrow{\text{CASO RECURSIVO}}$$

(en realidad en vez de otherwise deberia tener $m > 0$, para que no me acepte el -1 por ej.)

DONER CASO
DE ERROR
EN GUARDAS

otherwise = error "MAMAH!"

Como podría definir la función head?

head \rightarrow Devuelve el primer elemento de la lista

Lo defino por PATTERNMATCHING head :: [a] \rightarrow a
head (x : _) = x
(:)
LISTA

x seria el primer elemento

otro ej. (x : y : _) x PRIMER elem
y SEGUNDO elem

PATRON DE:

función ::
1 : [2,3] $\xrightarrow{\text{TODAS ME DAN}}$
1 : 2 : [3] $\xrightarrow{\text{[1,2,3]}}$
1 : 2 : 3 : []

(x : _) : LISTA MINIMO 1 elem
(x : y : _) : LISTA MIN 2 elem
(x) : LISTA DE SOLO 1 elem

ESTRUCTURA

LA LISTA ES RECURSIVA, la coda es una lista, que tiene como coda otra lista y Así...

$$[1, 2] == (1 : [2]) == (1 : 2 : [])$$

Es ListaVacia [] = TRUE

Es ListaVacia (cabecera; cola) = FALSE

o $\rightarrow (- : -)$ Si tiene cabecera y
cola no es Listavacia.

El paso base es el []

(Lista Vacia), es el momento
donde paro la recursividad

Sabiendo ESTO; si quiero definir la SUMATORIA de los elementos de una lista

SUMATORIA :: Num m \Rightarrow [m] \rightarrow m

SUMATORIA [] = 0

SUMATORIA (x; xs) = $x + \text{SUMATORIA } xs$
y
dista con
1 elemento
menos.

} En algún momento llegará
a la LISTA VACIA

sum [1, 2, 3]

1 + sum [2, 3]

1 + 2 + sum [3]

1 + 2 + 3 + sum []

1 + 2 + 3 + 0

y la concatenación.

CONCATENACION :: [[a]] \rightarrow [a]

Concatenación [] = []

Concatenación (x; xs) = $x ++ \text{Concatenación } xs$
 $(++) * (\text{concatenación } xs)$

y la conjunción.

CONJUNCION :: [Bool] \rightarrow Bool

Conjunción [] = TRUE

Conjunción (x; xs) = (x) \wedge Conjunción xs

puedo generalizarla en una función LLAMADA PLEGAR

función Plegar :: $(b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$
función Plegar operación valorInical [] = valorInical

función Plegar operación valorInical (x; xs) = operación \times (función Plegar operación valorInical xs)

Tengo que pasársela como parámetro la operación y el
VALOR INICIAL (semilla)

Ej: En SUMATORIA plegar (+) 0 NUMEROS

NOTA

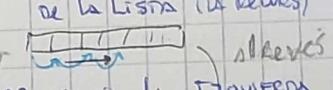
Aplicando a las funciones:

SUMATORIA :: Num m \Rightarrow [m] \rightarrow m
SUMATORIA numeros = plegar (+) @ numeros

y así...

La Función plegar vista anteriormente YA EXISTE!

y hay una familia de tipos, los FOLDABLE a los que se les puede aplicar.
El foldable que trabajaremos serán las LISAS

foldr :: ($b \rightarrow a \rightarrow a$) $\rightarrow a \rightarrow [b] \rightarrow a$ → Acceso x La Derecha
de la Lista (la recorres)

foldl :: ($a \rightarrow b \rightarrow a$) $\rightarrow a \rightarrow [b] \rightarrow a$ → Acceso x La Izquierda
de la Lista (la recorres)

19/5

```
data MATERIA = MATERIA {  
    CODIGO :: Int  
    NOMBREMATERIA :: String  
} deriving (Show)
```

```
data ESTUDIANTE = ESTUDIANTE {  
    LEGAJO :: Int,  
    NOMBREESTUDIANTE :: String  
    MATERIAS :: [MATERIA]  
} deriving (Show)
```

Fede = ESTUDIANTE 1231232 "Federico Scarpa" []
Juli = ESTUDIANTE 1597572 "Julian Bebel Alt" []

Algebra = MATERIA 084545 "AGA"
Analisis = MATERIA 084546 "Analisis I"
Syo = MATERIA 084547 "SIST. Y ORG"

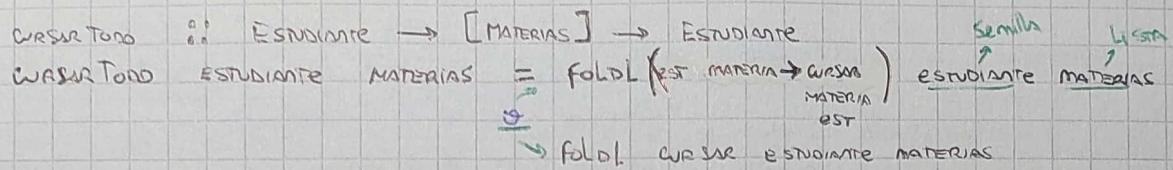
DEVUELVE EL ESTUDIANTE (UNO NUEVO PRACTICANDO A SUBIR)
CON UNA MATERIA +

CURSAR MATERIA (ESTUDIANTE Legajo nombre_materias) = ESTUDIANTE Legajo nombre (MATERIA: MATERIA)

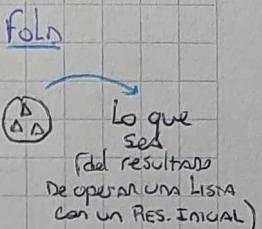
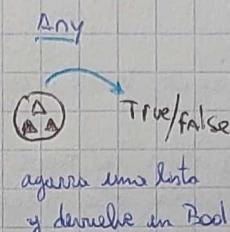
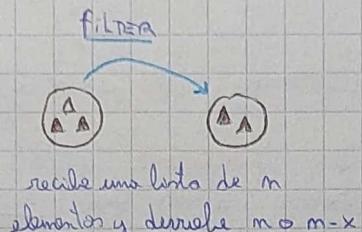
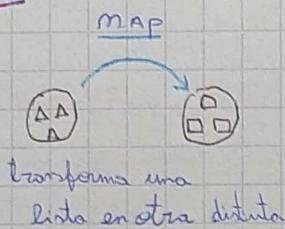
el tema con esta función es que si ya puse que cursa ALGEBRA, si quiero poner que CURSO SYO Me lo hace al viejo estudiante, y no al nuevo con AGA cursos.

NOTA CURSO SYO Me lo hace al viejo estudiante, y no al nuevo con AGA cursos.
Se puede solucionar así: CURSAR SYO (wesn AGA JULI) Pero hacer esto con tantas es ENGORRADO

haremos una nueva función CURSARTODO con FOLD



[NOTA]



[NOTA 2]

como existe la currificación, las siguientes notaciones son equivalentes:

$$\text{CURSAR_MATERIA} (\text{ESTUDIANTE } L \text{ n ms}) = \text{ESTUDIANTE } L \text{ n } (\text{MATERIA : ms})$$

Lo mismo

$$\text{CURSAR_MATERIA} = \backslash (\text{ESTUDIANTE } L \text{ n ms}) \rightarrow \text{ESTUDIANTE } L \text{ n } (\text{MATERIA : ms})$$

$$\text{CURSAR} = \backslash \text{MATERIA} \rightarrow \backslash (\text{ESTUDIANTE } L \text{ n ms}) \rightarrow \text{ESTUDIANTE } L \text{ n } (\text{MATERIA : ms})$$

$$\text{CURSAR} = \backslash \text{MATERIA } (\text{ESTUDIANTE } L \text{ n ms}) \rightarrow \text{ESTUDIANTE } L \text{ n } (\text{MATERIA : ms})$$

Si hiciera la misma función con RECURSIVIDAD:

$$\text{CURSARTODO } \text{ESTUDIANTE } [] = \text{ESTUDIANTE}$$

$$\text{CURSARTODO } \text{ESTUDIANTE } (m : ms) = \text{CURSARTODO } (\text{CURSAR } m \text{ ESTUDIANTE}) ms$$

pero es + complicada, por lo que es mejor haciendo con FOLD, es + declarativo

[NOTA]

LAZY EVALUATION

Las listas se podían definir también como:

$$[1, 2, 3, 4, 5] \rightarrow [1..5]$$

$$[1, 3 .. 10] \rightarrow [1, 3, 5, 7, 9]$$

y también **Listas Infinitas** $[1..] \rightarrow [1, 2, 3, \dots, \infty]$

CRD + PC
para parar

Lazy Evaluation es la forma que tiene Haskell de trabajar con sus valores

True || error "Esto es un error"

En lenguajes comunes se evalúa lo que está dentro de la función y luego el ||, en Haskell es al revés

take 10 [1..]

Primero evalúa el take 10 y luego la lista,
no viceversa

Por eso es **lazy evaluation, no evalúa de más.**

Unfold: Es el contrario de fold

• repeat 5 $\xrightarrow{\text{devuelve}}$ [5, 5, 5, ...] infinito

• iterate (+1) 5 $\xrightarrow{\text{devuelve}}$ [5, 6, 7, 8, ...] infinito

↓
iterar $:: (a \rightarrow a) \rightarrow a \rightarrow [a]$

iterar una función una semilla = una semilla : ITERAR UNA FUNCION (una función una Semilla)

no hay caso base porque la lista es ∞

• cycle [1, 2, 3] \longrightarrow [1, 2, 3, 1, 2, 3, 1, 2, 3, ...] infinito

• replicate 5 "holi" \longrightarrow ["holi", "holi", "holi", "holi", "holi"]

Doble X = X + X

Doble (doble 4)

call by value

doble (doble 4)

doble (4+4)

doble (8)

8 + 8

16

call by name

doble 4 + doble 4

4 + 4 + doble 4

8 + doble 4

8 + 4 + 4

12 + 4

16

Haskell hace call by name

con memory sharing

doble 4 + doble 4

4 + 4 + 4 + 4

8 + 8

16

que hace
que usa
expresión
se evalúa
solo 1 vez

⇒ Tenemos LAZY Evaluation porque nos da lo mismo evaluado en un momento o en otro, en funcional nada varía