Introducción a la Programación Funcional en Haskell

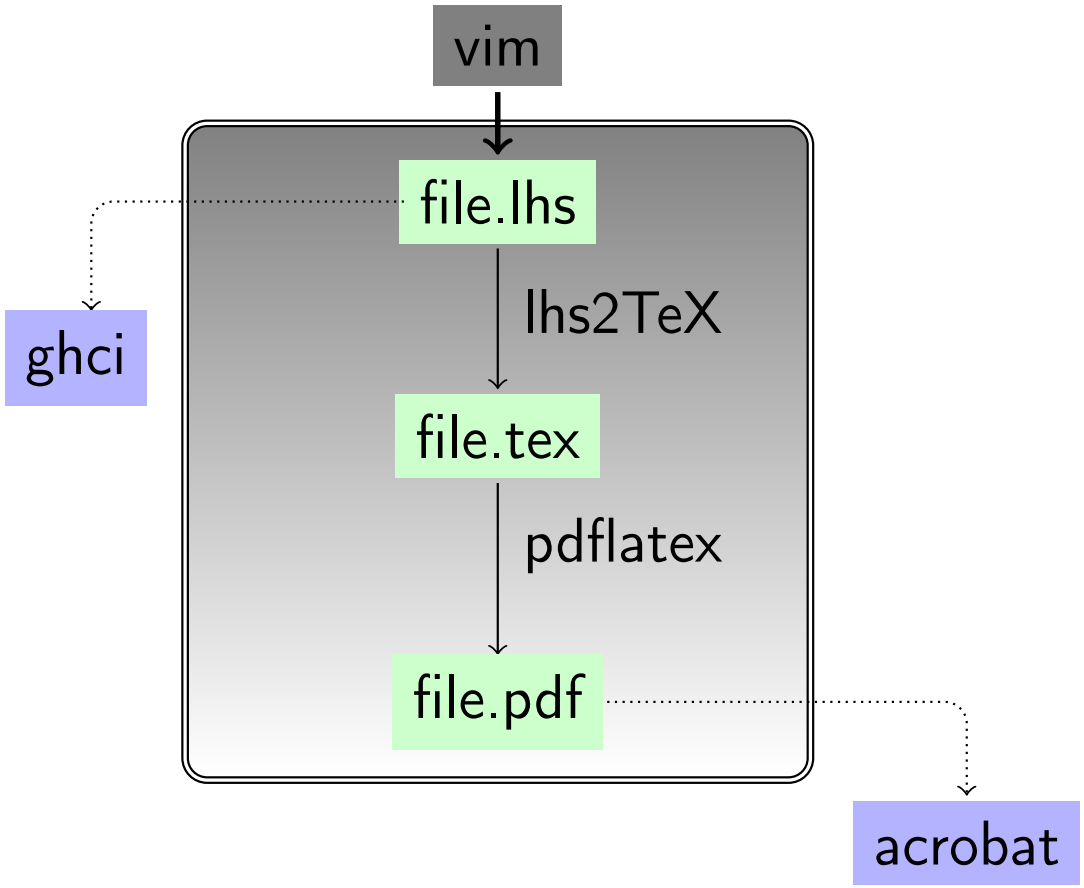
Gustavo Villavicencio[1]

[1]Facultad de Matemática Aplicada
Universidad Católica de Santiago del Estero

12 de Agosto, 2011

Ingeniería de documentos

Para entender el esquema de la presentación.



Outline

1. Preliminares
2. Primera Parte
   - Paradigmas de programación
   - Importancia de la programación funcional
   - Resumen
3. Segunda Parte
   - El intérprete Hugs
   - Funciones
   - Expresiones usuales
   - Ejercicios
   - Operadores de orden superior
   - Ejercicios
   - Resumen
4. Tercera Parte: Clases de tipos
   - El sistema de tipos en Haskell
   - Tipos de datos
   - Polimorfismo
   - Ejercicios
   - Resumen
5. Mónadas
   - El problema
   - Operaciones de I/O
   - Ejercicios

Programación imperativa

Programación imperativa

Los programas estás constituidos por sentencias (comandos) que alteran el estado del mismo. Ejemplos: C++, Java, etc.

Características importantes:

- En los programas imperativos el estado está compuesto por un conjunto de variables.
- El estado del programa cambia cuando alguna de las variables cambia. $\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_n = \sigma'$.
- Referential opaqueness - Efectos secundarios permitidos.
- Programa imperativo: Conjunto de sentencias sobre cómo realizar los cambios de estado.

Programación funcional

Características I

- El concepto de computación usado es el de función matemática. Esta se define como un mapeo de valores de un tipo a valores de otro tipo.
- La evaluación (ejecución) no es por transformación de la memoria sino por aplicación de funciones a los argumentos de entrada: Lambda calculus.

$$\begin{aligned} sumLista\ [\,] &= 0 \\ sumLista\ (h : t) &= h + sumLista\ t \end{aligned}$$

$$\begin{aligned} sumLista[7,5,9] &= sumLista(7 : [5,9]) \\ &= 7 + sumLista[5,9] \\ &= 7 + sumLista(5 : [9]) \\ &= 7 + 5 + sumLista[9] \\ &= 7 + 5 + sumLista(9 : []) \\ &= 7 + 5 + 9 + sumLista[] \\ &= 7 + 5 + 9 + 0 \\ &= 21 \end{aligned}$$

Programación funcional

Características II

- El comportamiento funcional se refuerza: $\sigma' = f(\sigma)$ (asumiendo programas determinísticos).
- Transparencia referencial - Los efectos secundarios no están permitidos. (Ejemplo en PresExamples.hs).
- Lazy evaluation (Haskell en particular).
- Ejemplos: Haskell, Scheme, etc.

Clasificación

- Puros: Haskell, Miranda, Clean
- Impuros: Standard ML, Scheme

Programación funcional

Otras características:

- Funciones como valores
- Funciones de orden superior
- Fuertemente tipado -> Inferencia de tipos
- Parametrización
- Pattern matching
- Non-strict functions -> Lazy evaluation (Haskell)

Areas de aplicación de Haskell

- Finanzas
- Comunicaciones (telefonía movil)
- Biotecnología
- IA
- Seguros
- Criptografía
- Diseño de hardware
- Tecnolología web

Por qué es importante la programación funcional?

Abstracción y Claridad

Los programas funcionales en Haskell resaltan el algoritmo evitando detalles de implementación. Como ya se dijo, enfatizan lo que tienen que realizar en lugar de cómo tienen que hacerlo.

$$
\begin{aligned}
&quicksort\ [\,] &&= [\,] \\
&quicksort\ (x : xs) &&= quicksort\ [e \mid e \leftarrow xs, e < x] +\!\!+ \\
&&&\quad [x] +\!\!+ \\
&&&\quad quicksort\ [e \mid e \leftarrow xs, e \geqslant x]
\end{aligned}
$$

Por qué es importante la programación funcional?

Abstracción y Claridad

Los programas funcionales en Haskell resaltan el algoritmo evitando detalles de implementación. Como ya se dijo, enfatizan lo que tienen que realizar en lugar de cómo tienen que hacerlo.

$$
\begin{aligned}
quicksort\ [\,] &= [\,] \\
quicksort\ (x : xs) &= quicksort\ [e \mid e \leftarrow xs, e < x] +\!\!+ \\
&\quad\ [x] +\!\!+ \\
&\quad\ quicksort\ [e \mid e \leftarrow xs, e \geqslant x]
\end{aligned}
$$

Abstracción y Claridad (Continuación)

Poseen mecanismos sofisticados de abstracción: Operadores de orden superior.

$$sumListaf = foldr\ (+)\ 0$$
$$prodListaf = foldr\ (*)\ 1$$

Por qué es importante la programación funcional?

Precisión

Los programas funcionales son apropiados para el razonamiento ecuacional gracias a la transparencia referencial.

Visión imperativa : Programas = Estructuras de datos + algoritmos

Visión funcional : Algebra = Objetos + operadores

Para observar esto, nos sumergimos en las "aguas profundas" de la programación funcional.

Por qué es importante la programación funcional?

Precisión

Los programas funcionales son apropiados para el razonamiento ecuacional gracias a la transparencia referencial.

Visión imperativa : Programas = Estructuras de datos + algoritmos
Visión funcional : Algebra = Objetos + operadores

Para observar esto, nos sumergimos en las "aguas profundas" de la programación funcional.

Por qué es importante la programación funcional?

Ejemplo de razonamiento ecuacional

Consideremos el tipo de datos listas en Haskell

data *Lista a* = *Nil* | *Cons a* (*Lista a*)

desde donde derivamos el functor: $\mathrm{F_A} : \underline{()} + \underline{A} \times I$

- Sobre tipos de datos

$$type\ \mathrm{F}_A\ B = () + A \times B$$

- Sobre funciones

$$\begin{array}{lcl} \mathrm{F}_A & :: & (A \longrightarrow B) \longrightarrow (\mathrm{F}_A\ A \longrightarrow \mathrm{F}_A\ B) \\ \mathrm{F}_A\ f & = & id + id \times f \end{array}$$

Por qué es importante la programación funcional?

Ejemplo de razonamiento ecuacional (continuación I)

Supongamos además dos F_A-algebras $(A, [Nil, Cons])$ and $(\mathrm{Z}, [\underline{0}, +])$ de la categoria **Alg**(F) de F-Algebras

- *Nil* es el operador de secuencia vacía: $Nil : 1 \longrightarrow L$.
- *Cons* es el constructor de secuencias: $Cons : A \times L \longrightarrow L$.
- $(A, [Nil, Cons])$ es un algebra inicial y $(\mathrm{Z}, [\underline{0}, +])$ es un algebra arbitraria.

Por qué es importante la programación funcional?

Ejemplo de razonamiento ecuacional (continuación II)

Homomorfismo: Traslación estructural entre objetos de una categoría.

$$\begin{array}{ccc} A & \xleftarrow{[Nil,Cons]} & 1 + Int \times A \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle id+id\times h} \\ \mathbb{Z} & \xleftarrow[[\underline{0},+]]{} & 1 + \mathbb{Z} \times \mathbb{Z} \end{array} \quad (1)$$

Por qué es importante la programación funcional?

Ejemplo de razonamiento ecuacional (continuación III)

Proceso de cálculo en notación pointfree. Notación pointfree?

$$dupAddPW\ x\ y = 2 * x + y \quad \text{-- Pointwise example}$$
$$dupAddPF = (+) \circ (2*) \quad \text{-- Pointfree example}$$

$$\begin{aligned} & h \cdot [Nil, Cons] && && (2) \\ & && = && \textit{from commutative diagram} \\ & [\underline{0}, +] \cdot (id + id \times h) && && \\ & && = && + - \textit{absorption} \\ & [\underline{0} \cdot id, + \cdot (id \times h)] && && \\ & && = && \textit{identity} \\ & [\underline{0}, + \cdot (id \times h)] && && \end{aligned}$$

Por qué es importante la programación funcional?

Ejemplo de razonamiento ecuacional (continuación IV)

Siguiendo con el proceso de cálculo reordenamos la ecuación

$$h \cdot [Nil, Cons] = [\underline{0}, + \cdot (id \times h)]$$

por la propiedad de fusión de la suma

$$[h \cdot Nil, h \cdot Cons] = [\underline{0}, + \cdot (id \times h)]$$

por igualdad estructural del operador

$$\begin{cases} h \cdot Nil & = & \underline{0} \\ h \cdot Cons & = & + \cdot (id \times h) \end{cases}$$

Porqué es importante la programación funcional

Ejemplo de razonamiento ecuacional (continuación V)

En notación pointwise las ecuaciones previas se reescriben como

$$\begin{cases} (h \cdot Nil)x & = & \underline{0}x \\ (h \cdot Cons)(a, xs) & = (& + \cdot (id \times h))(a, xs) \end{cases}$$

Aplicando definición de composición, de función constante, de producto de funciones y de suma

$$\begin{cases} h\ Nil & = & 0 \\ h\ Cons(a, xs) & = & a + h\ xs \end{cases}$$

Las ecuaciones finales corresponden a una función ejecutable en *HASKELL*!!!

Por qué es importante la programación funcional?

Capacidad de reuso

Gracias a los mecanismos algebraicos subyacentes en un lenguanje puramente funcional como Haskell, las posibilidades de reuso son significativas.

Recordemos (1). Qué ocurre si cambiamos la F_A-algebra $(\mathrm{Z}, [\underline{0}, +])$ por otra idéntica desde una perspectiva abstracta, es decir, isomórfica: $(\mathrm{N}, [\underline{1}, *])$

Por qué es importante la programación funcional?

Capacidad de reuso (continuación I)

Simplemente

$$\begin{cases} h\ Nil & = & 1 \\ h\ Cons(a, xs) & = & a * h\ xs \end{cases} \tag{3}$$

reusando todo el proceso de cálculo!
Más aún podemos reusar el esquema de recursión completo: Funciones de orden superior

$$h = foldr\ (*)\ 1$$

Por qué la PF no tiene un uso masivo?

Causas:

- Complicado de aprender y/o aplicar. ✖
- Beneficios poco claros o no significativos respecto de la PI ✔
- Resistencia al cambio ✔

Por qué la PF no tiene un uso masivo?

Causas:

- Complicado de aprender y/o aplicar. ✖
- Beneficios poco claros o no significativos respecto de la PI ✔
- Resistencia al cambio ✔

Por qué la PF no tiene un uso masivo?

Causas:

- Complicado de aprender y/o aplicar. ✖
- Beneficios poco claros o no significativos respecto de la PI ✔
- Resistencia al cambio ✔

Por qué la PF no tiene un uso masivo?

Causas:

- Complicado de aprender y/o aplicar. ✖
- Beneficios poco claros o no significativos respecto de la PI ✔
- Resistencia al cambio ✔

Resumen

A destacar de la programación funcional:

- Abstracción, ... programas fáciles de entender
- reuso y
- robustez. Razonamiento ecuacional:
  - Construcción y
  - transformación de programas.

Resumen

A destacar de la programación funcional:

- Abstracción, ... programas fáciles de entender
- reuso y
- robustez. Razonamiento ecuacional:
  - Construcción y
  - transformación de programas.

Fin de la primera parte

Tipos de archivos

- .hs: Haskell Script: Conjunto de definiciones y declaraciones.
- .lhs: Literate Haskell Script: Texto + código Haskell.
  Nota: Recordar 2.

Módulos

agrupar funcionalidades relacionadas.

- Encapsula colecciones de valores, tipos de datos, clases, etc.
- Exportan recursos para hacerlos accesibles a otros módulos.
- Importan recursos de otros módulos.

Definición de módulos

- Un módulo por archivo
- Primera letra del nombre con mayúscula

El intérprete Hugs

Sesión típica

- Leer expresión
- Evaluar expresión
- Imprimir resultado

de sesión típica

```
gustavo@linux-tqkb:~/Projects/OnGoing/Talks> hugs -98
__   __ __  __  ____   ___      ------------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ------------------------------------------

Hugs mode: Restart with command line option +98 for Haskell 98 mode

Type :? for help
Hugs> 3+6
9
Hugs> 'a' < 'x'
True
Hugs> sqrt 16
4.0
Hugs> print "-98 option enables some special Hugs extensions"
"-98 option enables some special Hugs extensions"
```

El intérprete Hugs

```
Hugs> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:edit <filename>    edit file
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names exported by <modules>
:main <aruments>    run the main function with the given arguments
:find <name>        edit module containing definition of name
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
Hugs>
```

Comandos más usados

- Setea el editor de texto

```
Hugs>:s -Egvim
```

- Carga un archivo

```
Hugs>:l PresExamples.hs
```

- Recarga el último archivo cargado

```
Hugs>:r
```

- Mostrar el tipo de una función

```
Main> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
Main>
```

Algunos operadores de Haskell

Cuadro : Operadores Haskell

–	Inicio línea de comentario
{-	Inicio comentario corto
-}	Fin comentario corto
^, ^^, **	Potenciación
&&	**And**
||	**Or**
==	Igual
/ =	No igual
\	Operador lambda
|	Separador en definición de listas por comprensión
	Alternativa en definición de tipos de datos
++	Concatenación de listas
!!	Indexación de listas

Creación de un módulo

Luego de setear el editor que se usará, crear el módulo **MiModulo.hs mediante**

```
Hugs> :e MiModulo.hs
Hugs>
```

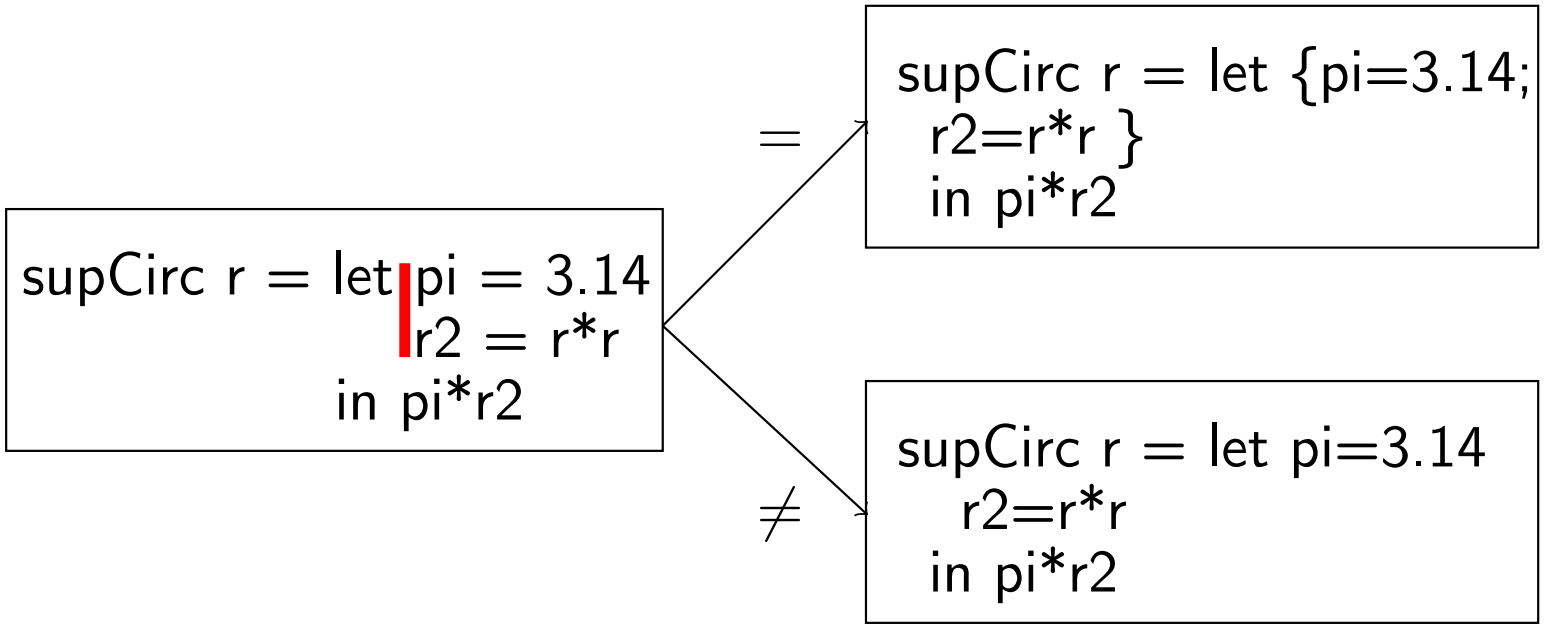
y como cabecera del módulo introducir

```
modulo MiModulo where
```

Haskell layout

- Haskell usa la indentación para establecer el final de definiciones, expresiones, etc.
- Propósito: "Limpiar" el código de caracteres innecesarios.
- Regla básica general: Una definición termina cuando un caracter no vacío, aparece en la misma columna como el primer símbolo de la definición.

Haskell layout example



Haskell layout example

Prueba las siguientes versiones de la función *cubo* ingresándolas en **MiModulo.hs**

$cubo\ x = x * x * x$ -- **correct version**

Versiones incorrectas

```
cubo x = x *
x * x

cubo x
= x * x * x
```

Concepto

Es un mapeo de valores de un tipo a valores de otro tipo.

Recuerda: **Haskell es un lenguaje puramente funcional!**

Función pura

El resultado depende únicamente de los datos de entrada y no genera *efectos secundarios*. El resultado NO depende de cambios del estado del programa ni de entradas de fuentes externas.

Ejemplo: Función que transforma centímetros en pulgadas.

$$cenPul\ c = c * 2{,}54$$

Función impura

Puede generar resultados según el estado externo, o bien, puede modificar el estado externo.

Ejemplo: Función que toma como entrada un string desde el teclado.

Consideraciones sobre funciones

- Sin embargo, para modelar la realidad son necesarias tanto las funciones puras como las impuras.
- Pero si Haskell es puramente funcional, cómo modela las funciones impuras?

Mónadas

Consideraciones sobre funciones

- Sin embargo, para modelar la realidad son necesarias tanto las funciones puras como las impuras.

- Pero si Haskell es puramente funcional, cómo modela las funciones impuras?

Mónadas

Consideraciones sobre funciones

- Sin embargo, para modelar la realidad son necesarias tanto las funciones puras como las impuras.

- Pero si Haskell es puramente funcional, cómo modela las funciones impuras?

Mónadas

Consideraciones II

- Por otro lado, las funciones son valores, por lo que pueden ser transferidas a otras funciones.
- Las funciones que reciben otras funciones como argumentos se denominan funciones de orden superior.
- Funciones anónimas: no se indica el nombre de la función solo el cuerpo de la misma.
- Y almacenadas en estructuras de datos!

$$listFunc = [\lambda x\ y \rightarrow x + y, \lambda x\ y \rightarrow x - y, \lambda x\ y \rightarrow x * y, \lambda x\ y \rightarrow x\ `div`\ y]$$

Probar en hugs

```
Main> (listFunc!!(1)) 4 5
```

Consideraciones III

En Haskell las funciones son no estrictas.

Probar:

```
valor = 1 / 0   -- Note: It is a definition, not a computation.
constante x = 1
```

Recursión

En Haskell, las funciones pueden ser definidas en términos de ellas mismas.

$$facR\ 0 = 1$$
$$facR\ n = n * facR\ (n-1)$$

También tenemos versiones no recursivas

$$facNR\ n = product\ [1\,..\,n]$$

Ventajas de la recursión

- Muchas funciones son definidas naturalmente en téminos de ellas mismas

```
sum []     = 0
sum (h:t) = h + sum t
```

Ventajas de la recursión

- Muchas propiedades sobre funciones recursivas pueden demostrarse aplicando inducción matemática

```
map f (xs ++ yx) = map f xs ++ map f ys
```

Definición de funciones

- En algebra una función se expresa como $f : A \longrightarrow B$, donde A es el dominio y B el codomio o contradominio.
- De igual modo, en Haskell escribimos:

```
increment :: Int → Int   -- try comment this line
increment n = n + 1
```

donde el *Int* a la izquierda del operador de tipo de función (->) es la fuente de la función, en tanto que el de la derecha es el rango de la función.

- La expresión a la derecha del símbolo :: es el tipo de la función.

A cerca del tipo de una función

- Recuerda! No es obligatorio explicitar el tipo de una función.
- El tipo de una función puede tener múltiples argumentos. El operador de tipo de función $(->)$ es asociativo a derecha.
  Probar con y sin paréntesis la siguiente función

$$concatena :: String \to (String \to String)$$
$$concatena\ s1\ s2 = s1 +\!\!+ s2$$

A cerca del tipo de una función (continuación)

Currying

- Asimismo, la aplicación de funciones es asociativo a izquierda: *concatena s1* es una función de tipo **String -> String** (partial application).
  Ingresa la siguiente función en **MiModulo.hs**

```
partialConcat = concatena "mundo"   -- new function with one argument
```

  Recarga **MiModulo.hs** e intenta en hugs

```
Main> partialConcat "loco"
```

- La aplicación parcial de funciones de denomina currying

A cerca del tipo de una función (continuación)

- La versión *uncurried* de la función **concatena** sería

$$uconcatena :: (String, String) \rightarrow String$$
$$uconcatena\ (s1, s2) = s1 +\!\!+ s2$$

- *Operadores infijos* como (++) son realmente funciones por lo que pueden ser parcialmente aplicados. A esto se denomina section

Prueba en hugs

```
Main> ("mundos " ++) "loco"   -- Remember: ("mundos " ++) is a function
```

Composición de funciones

Al igual que en algebra, cuando el codominio de una función coincide con el dominio de otra, las mismas pueden ser compuestas

$$wordCount = length \circ words \quad \text{-- both function in Prelude}$$

Pattern Matching

Se trata de un mecanismo sintáctico que permite clarificar la definición de funciones cotejando sus argumentos con patrones.

$$primero\ (x, y) = x$$
$$segundo\ (x, y) = y$$
$$prim\ (x, \_) = x$$
$$segu\ (\_, y) = y$$

_ es un comodín.

Pattern matching sobre listas

- El operador (:) es el constructor de listas mientras que ([]) denota lista vacía
  Prueba en Hugs

```
Main> (:) 4 [3,7]
```

- El operador (:) es usado en los patrones sobre listas para descomponer las mismas

$$head :: [a] \rightarrow a \quad \text{-- In Prelude}$$
$$head\ (h : t) = h$$

- En tanto que ([]) es usado para detectar listas vacías

$$sumList :: Num\ a \Rightarrow [a] \rightarrow a$$
$$sumList\ [\,] \quad = 0$$
$$sumList\ (h : t) = h + sum\ t$$

Pattern matching sobre listas

- El operador (:) es el constructor de listas mientras que ([]) denota lista vacía
  Prueba en Hugs

```
Main> (:) 4 [3,7]
```

- El operador (:) es usado en los patrones sobre listas para descomponer las mismas

$$head :: [a] \to a \quad \text{-- In Prelude}$$
$$head\ (h : t) = h$$

- En tanto que ([]) es usado para detectar listas vacías

$$sumList :: Num\ a \Rightarrow [a] \to a$$
$$sumList\ [\,] = 0$$
$$sumList\ (h : t) = h + sum\ t$$

Pattern matching sobre listas

- El operador (:) es el constructor de listas mientras que ([]) denota lista vacía
  Prueba en Hugs

```
Main> (:) 4 [3,7]
```

- El operador (:) es usado en los patrones sobre listas para descomponer las mismas

$$head :: [a] \to a \quad \text{-- In Prelude}$$
$$head\ (h : t) = h$$

- En tanto que ([]) es usado para detectar listas vacías

$$sumList :: Num\ a \Rightarrow [a] \to a$$
$$sumList\ [\,] = 0$$
$$sumList\ (h : t) = h + sum\ t$$

Ejercicios

1. Definir una función recursiva que aplique las funciones almacenadas en *listFunc* a un par de argumentos.
2. Ingresar la siguiente lista en **MiModulo.hs**

   ```
   mundos =  [" lindo ", " feo ", " loco "]
   ```

   Usando la función *partialConcat*, definir una función recursiva que genere como salida:

   ```
   ["mundo lindo", "mundo feo", "mundo loco"]
   ```

3. Genera otra versión de la función anterior, usando aplicación parcial sobre el operador de concatenación de listas (++).

Guardias

Se trata de un mecanismo sintáctico que facilita el entendimiento de funciones que usan múltiples condiciones

```
mm n m | n > m     = "El primero es mayor"
       | n < m     = "El segundo es mayor"
       | otherwise = "Ambos son iguales"
```

Listas por comprensión

Se trata de una notación sintáctica originada en la definición de conjuntos por comprensión que permite la generación de listas a partir de otras existentes.

$$cubo10 = [x3 \mid x \leftarrow [1\,..\,10]]$$

Notas I:

- x <- [1..10] es el generador
- Puede haber múltiples generadores separados por comas

Listas por comprensión

Se trata de una notación sintáctica originada en la definición de conjuntos por comprensión que permite la generación de listas a partir de otras existentes.

$$cubo10 = [x3 \mid x \leftarrow [1\,..\,10]]$$

Notas I:

- x <- [1..10] es el generador
- Puede haber múltiples generadores separados por comas

Notas II:

- Alterando el orden de los generadores se altera el orden de los elementos en la lista final

$$pares1 = [(x, y) \mid x \leftarrow [1..3], y \leftarrow [4, 5]]$$

No es lo mismo que

$$pares2 = [(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1..3]]$$

- Múltiples generadores funcionan como loops anidados

Notas II:

- Alterando el orden de los generadores se altera el orden de los elementos en la lista final

$$pares1 = [(x, y) \mid x \leftarrow [1..3], y \leftarrow [4, 5]]$$

No es lo mismo que

$$pares2 = [(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1..3]]$$

- Múltiples generadores funcionan como loops anidados

III

- Generadores dependientes: generadores definidos al final pueden depender de los definidos al inicio

$$gd = [(x, y) \mid x \leftarrow [1..3], y \leftarrow [x..4]]$$

- Se pueden filtrar los elementos producidos por los generadores usando guardias

$$factores\ n = [x \mid x \leftarrow [1..n], n\ `mod`\ x \equiv 0]$$

case ... of ...

Se trata de un **if ... then ...else ...** generalizado que evalúa una condición para determinar la expresión a evaluar

Nota I:

- **if...then...else** generalizado

```
if e then e1 else e2
```

≡

```
case e of
    True -> e1
    False -> e1
```

- Los tipos de las expresiones de los miembros derechos deben ser los mismos

case ... of ...

Se trata de un **if ... then ...else ...** generalizado que evalúa una condición para determinar la expresión a evaluar

Nota I:

- **if...then...else** generalizado

```
if e then e1 else e2
```

≡

```
case e of
    True -> e1
    False -> e1
```

- Los tipos de las expresiones de los miembros derechos deben ser los mismos

II:

- La condición a evaluar no necesariamente debe ser de tipo **Bool** como en el **if...then...else**. La condición puede hacer referencia a un patrón.

$$
\begin{aligned}
&\mathit{suml}\ l = \textbf{case}\ l\ \textbf{of} \\
&\qquad\qquad [\,] \quad\ \ \to 0 \\
&\qquad\qquad (h : t) \to h + \mathit{suml}\ l
\end{aligned}
$$

Declaraciones locales: Expresiones **let ...in ...**

Las expresiones **let...in...** permiten realizar declaraciones locales a una función

$$\begin{aligned} \mathit{superfCirc}\ \mathit{radio} = \textbf{let}\ & \mathit{pi} = 3{,}14 \\ & \mathit{radio2} = \mathit{radio} * \mathit{radio} \\ \textbf{in}\ & \mathit{pi} * \mathit{radio2} \end{aligned}$$

Notas:

- También está permitido definir funciones de manera local
- Las definiciones solo pueden ser "vistas" por la función donde se encuentran

Expresiones **where**

La cláusula **where** es similar a **let...in...** pero difieren en la posición de las definiciones locales.

```
supCircW radio = pi * radio2
                 where pi = 3,14
                       radio2 = radio * radio
```

1. Usando la notación de listas por comprensión, escribir una función que dada una condición y una lista, filtre los elementos que cumplan la condición. Por ejemplo

```
filtro (>3) [5,1,7,0,9] = [5,7,9]
```

2. Dado un número y una lista ordenada, insertar el número en la posición correcta.
3. Explique el funcionamiento de la siguiente función

```
funDesc ll = [x | l1 <- ll, x <- l1]
```

4. Escribir una función recursiva para multiplicar dos enteros usando adición.

1. Transforma la función *suml*

```
suml []     = 0
suml (h:t) = h + suml t
```

   aplicando la expresión **let...in...**

2. Escribe una función que acepte dos strings y cuente la cantidad de veces que aparecen los caracteres que componen el primer string, en el segundo. Por ejemplo:

```
cuentaC "az" "bvvvatsszaaaz" = 6
```

3. Escribe una versión alternativa de la función anterior.

foldr

Ya hemos visto en 19 un esquema de recursión típico:

Suma y producto

$$
\begin{aligned}
&sumLista2\ [\,] &&= 0\\
&sumLista2\ (h : t) &&= h + sumLista2\ t\\
&sumListaf2 &&= foldr\ (+)\ 0\\
&prodLista\ [\,] &&= 1\\
&prodLista\ (h : t) &&= h * prodLista\ t\\
&prodListaf2 &&= foldr\ (*)\ 1
\end{aligned}
$$

El esquema de recursión que muestra *sumLista2* y *prodLista* está encapsulado en **foldr**.

foldr en detalle

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (h:t) = f h (foldr f z t)
```

Prueba en Hugs

```
Main>  foldr (\x y -> (x+y)/2) 0 [11,2,5]
6.625
Main>
```

Observando la definición del operador **foldr**, podrías explicar el resultado obtenido mediante una secuencia de reducciones? (Recordar ejemplo anterior 5).

Función **map**

Aplica una función pasada como argumento a los elementos de una lista

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (h:t) = f h : map f t
```

Cómo duplicarías los elementos de una lista usando esta función?

- Dada una lista, cuyos elementos son a su vez listas, detectar las listas elementos cuyas longitudes superen un valor fijado. Escribe dos versiones de la función, una usando recursión explícita y otra usando la función **filter** del preludio.
- La función **concat** en el Prelude concatena en una lista, las listas que constituyen una lista dada como argumento (lista de listas)

```
concat :: [[a]] -> [a]
concat [[3,4,1],[1],[8,4]] = [3,4,1,1,8,4]
```

  Escribe una versión propia de **concat** usando **foldr**.
- Prueba el funcionamiento de la función **takeWhile** del Prelude. Escribe una versión propia de esta función aplicando recursión explícita y otra usando el operador **foldr**.

Resumen de la segunda parte

Características a destacar:

- Recursión
- Pattern matching
- Inferencia de tipos

Fin de la segunda parte

Generalidades

- Estáticamente tipado: errores detectados en tiempo de compilación.
- Los tipos pueden ser inferidos: Haskell no fuerza a indicar el tipo de una expresión.
- Las funciones son valores, por lo tanto, tienen un tipo asociado.

Tipos

- Básicos: Int, Bool, Char, etc
- Tipos predefinidos: Maybe y [a]

```
Maybe a  =  Nothing  |  Just a

[a] = [] | (:) a [a]
```

  - *Maybe* es un constructor de tipo, en tanto que *Nothing* y *Just* son constructores de datos de tipo *Maybe*.
  - *[a]* es un constructor de tipo, en tanto que *[]* y *(:)* son constructores de datos de tipo *[a]*.

Prueba en Hugs

```
Main> (:) 3 ((:) 4 ((:) 5 []))
```

Escribe la expresión anterior en notación infija.

Las cláusulas *data* y *type*

- *data*: Introduce nuevos tipos de datos

  data *Color* = *Rojo* | *Azul* | *Blanco* -- enumerated type example

- *type*: Renombra tipos existentes

  type *Nombre* = *String*

Tipos de datos algebraicos

constructores de datos pueden tener valores asociados

data *Color2* = *Verde* | *Negro* | *EscalaGris Int*

- Los valores del tipo de dato definido son valores de tipos de datos "encapsulados" en los constructores del tipo
- La única forma de acceder a los datos es "desencapsularlos" mediante pattern matching

$$\begin{aligned} isGreen\ Verde &= True \\ isGreen\ \_ &= False \end{aligned}$$

- Casos particulares de tipos algebraicos son los tipos enumerados, los tipos sumas, y los tipos productos o tuplas

Tipos recursivos

Recordar nuestra propia definición de listas en Haskell mostrada en 12. O bien

```
data ArbolBin a = Hoja a | Nodo (ArbolBin a) a (ArbolBin a)
                  deriving (Eq, Show)
arbolEjemplo = Nodo (Nodo (Hoja 7) 1 (Hoja 6)) 5 (Nodo (Hoja 11) 15 (Hoja 4))
```

Los valores construidos mediante el constructor de datos **Nodo**, contienen valores del mismo tipo que se está definiendo.

Tipos recursivos

: Sobre la cláusula **deriving**

- Cómo compararías en Hugs valores de tipo *ArbolBin a*?
- Comenta la línea en donde se encuentra la cláusula **deriving ...** y prueba comparar los valores anteriores de nuevo. Qué problemas observas?

Mediante la cláusula **deriving** el sistema genera automáticamente instancias de los métodos definidos en las clases para el tipo que se está definiendo:

- $(==)$ y $(/=)$ funciones en la clase **Eq**: Se generan instancias de estas funciones para *ArbolBin a*
- Funciones de impresión para *ArbolBin a* mediante los métodos de la clase **Show**

Tipos recursivos

: Sobre la cláusula **deriving**

- Cómo compararías en Hugs valores de tipo *ArbolBin a*?
- Comenta la línea en donde se encuentra la cláusula **deriving ...** y prueba comparar los valores anteriores de nuevo. Qué problemas observas?

Mediante la cláusula **deriving** el sistema genera automáticamente instancias de los métodos definidos en las clases para el tipo que se está definiendo:

- $(==)$ y $(/=)$ funciones en la clase **Eq**: Se generan instancias de estas funciones para *ArbolBin a*
- Funciones de impresión para *ArbolBin a* mediante los métodos de la clase **Show**

Tipos parametrizados

Polimorfismo paramétrico

- Los tipos de datos pueden estar parametrizados. Recordar

```
data Lista a    = ...
data ArbolBin a = ...
```

  a es una variable de tipo.

- **Lista** y **ArbolBin** son tipos polimórficos
- Recordar el comportamiento de la función **length**

No importa el tipo del argumento, la función se comporta siempre del mismo modo.

Clases de tipos

Notar que en **hugs** podemos hacer

```
Hugs> 'c' == 's'
False
Hugs> 2 == 6
False
Hugs>  "hi" == "hola"
False
```

El operador (==) está sobrecargado, puede operar sobre varios tipos de datos. De igual modo, (+) puede operar sobre varios tipos numéricos.

- El comportamiento de los operadores está diferenciado para cada tipo.
- A diferencia del polimorfismo paramétrico, el tipo determina el comportamiento de la función.
- Haskell permite organizar esta clase de polimorfismo, mediante clases de tipos.

Clases de tipos

Polimorfismo adhoc(overloading)

- A diferencia del polimorfismo parámetrico, las funciones que responden a este tipo de polimorfismo, no tienen un comportamiento uniforme, sino que ajustan el mismo según el tipo que las parametriza. También se denominan funciones sobrecargadas. Por ejemplo $+$.
- Las clases de tipos constituyen el mecanismo através del cual Haskell soporta polimorfismo adhoc.

Clases de tipos

Polimorfismo adhoc(overloading): Ejemplo I

$$\begin{aligned} isInList\ x\ [\,] &= False \\ isInList\ x\ (h:t) &= x \equiv h \vee isInList\ x\ t \end{aligned}$$

- Cuál sería el tipo tentativo de la función?
- Qué ocurre si queremos comparar los elementos de la lista *listFunc*?

El operador (==) no puede ser aplicado a todos los tipos

Clases de tipos

Polimorfismo adhoc(overloading): Ejemplo I

$$\begin{aligned} isInList\ x\ [\,] &= False \\ isInList\ x\ (h:t) &= x \equiv h \vee isInList\ x\ t \end{aligned}$$

- Cuál sería el tipo tentativo de la función?
- Qué ocurre si queremos comparar los elementos de la lista *listFunc*?

El operador (==) no puede ser aplicado a todos los tipos

Clases de tipos

Polimorfismo adhoc(overloading): Ejemplo I

Para especificar los tipos sobre los que es posible aplicar (==)

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Integer where
  x==y  = x `integerEq` y

instance Eq Float where
  x==y  =  x `floatEq` y
```

En particular, en el ejemplo previo 71, Haskell deriva automáticamente las instancias correspondientes para el tipo definido.

Clases de tipos

Polimorfismo adhoc(overloading): Ejemplo II

Supongamos el siguiente tipo de datos

```
data Expresion = Constante Int
               | Suma Expresion Expresion
               | Producto Expresion Expresion
               deriving Eq
```

y deseamos comparar datos de este tipo

```
-- instance Eq Expresion where
-- Constante a == Constante b = a == b
-- Suma e1 e2 == Suma e3 e4 = e1 == e3  e2 == e4
-- Producto e1 e2 == Producto e3 e4 = e1 == e3  e2 == e4
```

Ejercicios I

1. Definir un tipo de datos con los días de la semana. Luego definir la clase **Laborable** con un método que determine si un día es laborable o no. Finalmente definir una instancia de dicha clase.
2. Sea el siguiente tipo de datos

```
data ListaInt    = Nula | CLI Int ListaInt
```

y la clase **Estructuras** definida como

```
class Estructuras c where
  insertarElem  :: c -> Int -> c
  esMiembro     :: c -> Int -> Bool
```

Hacer al typo **ListaInt** una instancia de la clase **Estructuras**.

Resumen de la tercera parte

Características a destacar:

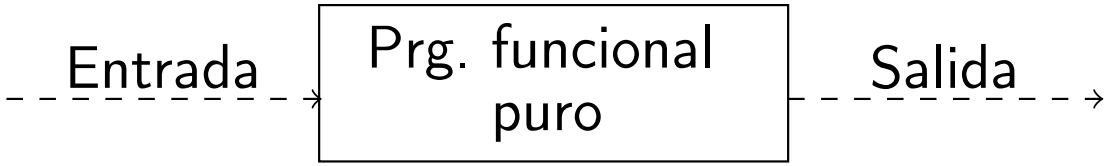
- Polimorfismo paramétrico
- Polimorfismo adhoc (overloading)

Fin de la tercera parte

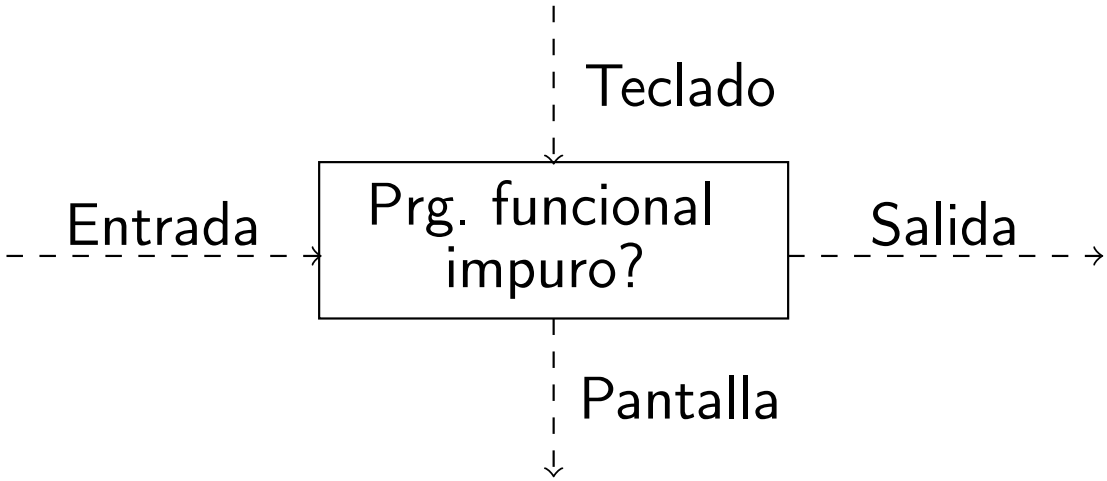
Algunas notas preliminares sobre mónadas

- La bella y la bestia:
  - Programación funcional pura: capacidad de abstracción (polimorfismo paramétrico y adhoc, funciones de orden superior, tipos de datos algebraicos) y similitud matemática (razonamiento ecuacional, las funciones son funciones matemáticas, independencia en el orden de evaluación).
  - Efectos secundarios: manejo de errores, operaciones de I/O, interface con otros lenguajes.
- No hay un modo único de explicar mónadas y que sea útil para todos.

Hasta ahora, teníamos programas Haskell que se evaluaban en modo batch



Pero obvia y necesariamente, también podemos tener un esquema interactivo



Recuerda

Un programa Haskell es una función matemática pura.

Por lo tanto, que ocurre con operaciones como

- Leer un carácter desde el teclado.

que no es una función matemática pura?
O bien,

- Imprimir un carácter en la pantalla.

que tampoco es una función matemática.

ejemplo del problema

-- Lista con dos funciones como elementos
$dosOps = [3 + 5, 2 * 7]$

-- Lista con dos ACCIONES como elementos
$dosCaracteres = [getChar, getChar]$

Dos problemas se manifiestan en el ejemplo:

- `getchar` se evalúa solamente una vez?: `[a,a]`.
- Si hay dos llamadas distintas a `getchar`, el orden de la evaluación no está garantizado: `[a,b]` o `[b,a]`.

ejemplo del problema

$$dosStrings = [putStr\ \texttt{"hola "}, putStr\ \texttt{"mundo"}]$$

donde

```
putStr :: String -> IO ()
```

putStr recibe un string y lo imprime en la pantalla. Pero esto no ocurre en la función *dosStrings* porque no hay acceso a los valores devueltos por *putStr*. Solo el compilador tiene acceso a ellos!

La mónada IO

Sobre el tipo de dato

```
type IO a = world -> (a,world)
```

Notas:

- **IO** es un constructor de tipo.
- **IO a** es una expresión que tiene como valor una **computación**, "que cuando se ejecuta", realiza una operación de **I/O** y devuelve un valor de tipo **a**.
- Una expresión con tipo **IO a** puede realizar alguna **acción** como consecuencia de su ejecución, antes de devolver un valor de tipo **a**.
- Evaluar una acción no tiene efectos, ejecutarla si.

La mónada IO

Sobre el argumento **world**:

- Determina el orden de ejecución de las acciones.
- Es transparente para el usuario.

```
-- secuencia :: [IO ()]
secuencia []     = return []
secuencia (h:t) = h >>= λx → secuencia t >>= λr → return (x:r)

secuenciaDo []     = return []
secuenciaDo (h:t) = do r1 ← h
                       r2 ← secuenciaDo t
                       return (r1:r2)
```

La mónada IO

el operador »= (bind):

- Las acciones solo pueden combinarse mediante este operador.
- El operador fuerza la secuencialidad:

```
a >>= \x -> b
```

  - Se ejecuta la acción **a** generando un resultado **r**.
  - Se aplica la función del lado derecho del operador »= a **r**.
  - Se ejecuta la acción resultante.
  - Se devuelve el valor resultante.

La mónada IO

el combinador **return**

$$dosChars = getChar \gg\!= \lambda c1 \rightarrow getChar \gg\!= \lambda c2 \rightarrow return\ (c1, c2)$$

return no realiza ninguna acción, simplemente devuelve un valor.
Algunas operaciones I/O:

- `putChar :: Char -> IO ()`
- `putStr :: String -> IO ()`
- `putStr :: String -> IO ()`
- `getChar :: IO Char`
- `getLine :: IO String`

La notación **do**

ejemplo previo en notación **do**

$$\begin{aligned} dosCharsDo = \textbf{do}\ & c1 \leftarrow getChar \\ & c2 \leftarrow getChar \\ & return\ (c1, c2) \end{aligned}$$

- Hace que el código parezca más imperativo.
- El alcance de las variables vinculadas en un generador es el resto de la expresión **do**.
- El última expresión en un bloque **do** debe ser una acción.
- **return** no necesariamente debe ser la última expresión.

Aspectos comunes

Los aspectos que las mónadas describen comparten operaciones comunes.

```
class Monad a where
  return :: a -> m a
  >>=    :: m a -> (a -> m b) -> m b
```

Notas:

- **return**: función polimórfica.
- »= (operador bind): permite la combinación de mónadas.

Estas funciones son una suerte de interface funcional de la mónada.

Ejercicios

1. Escribe tu propia versión de la función **getLine** aplicando la función (también en las bibliotecas estárdars de Haskell) **getChar**.
2. Define un tipo de datos que devuelva el valor apropiado si una operación tiene éxito, o un valor específico si se produce alguna falla. Aplica el tipo definido en una función que divida dos números, generando el valor de falla en el caso de división por cero.
3. Dados tres valores del tipo de dato definido en el ítem anterior, divide el primero por el segundo y el resultado por el tercero. Genera la falla correspondiente en el caso de división por cero. Escribe dos versiones de esta función, una usando la notación tradicional de mónadas y la otra usando la notación **do**.
4. Dada una lista de valores del tipo definido en el ítem 2 sumar los valores de la lista. Devolver una falla en el caso que alguno de los valores sea cero.
5. Al tipo de datos definido en el ítem 2, podrías definirlo como una instancia de la clase **Monad**?

Fin