MURAL

Capítulo 1: Paradigma Funcional

1. Función

Según el paradigma **funcional**, cada operación es una **función** (es decir, una relación entre un *dominio* y una *imagen*).

Estas relaciones, cumplen tanto la propiedad de *existencia* (para cada entrada tenemos una salida) como la de *unicidad* (todas las salidas son únicas).

1.1 Cómo interpretamos a las funciones

Las funciones pueden ser pensadas como un proceso de **caja negra**, el cual tiene sus respectivas entradas que se *transforman* en salidas (como es de caja negra, se ignora el cómo se transforman).

A su vez, también se pueden interpretar como funciones matemáticas que transforman valores.

En ambos casos, se exhibe la transparencia referencial ya que las funciones son *determinísticas* para los mismos parámetros. A su vez, no producen **efecto**, es decir, no pueden mutar sus argumentos ni otras variables (por ende, este paradigma no tiene *asignación destructiva*).

En este paradigma, las funciones también se conciben como un **TAD**. Su operación primitiva es la *aplicación* (es decir, aplicar una función a un valor).

Más específicamente en Haskell, la operación de aplicación se representa mediante el operador **($)**, el cual es una función **infija** que toma una función y la aplica a un valor.

```
funcionUno:: Number -> Number
funcionUno numero = sumarUno (sumarUno (doble numero))

funcionDos:: Number -> Number
funcionDos numero = sumarUno $ sumarUno $ doble numero
```

Como podemos observar **funcionUno** se encarga de sumarle uno al resultado de evaluar “sumarUno (doble numero)” que a su vez, suma uno al resultado del doble de un número.

Los paréntesis se ponen justamente para que la función, reciba como parámetro el resultado de evaluar otra función. La aplicación $, justamente se encarga de hacer eso y su prototipo es:

```
($) :: (a -> b) -> a -> b
($) funcion parametro = funcion parametro
```

1.2 Características de las funciones en Haskell

Las funciones en Haskell presentan todo lo anteriormente mencionado y lo siguiente:

- Las funciones son **valores**
- Las funciones tienen un **tipo** (*Dom → Img*)
- Todas las funciones están **currificadas** (tienen un solo argumento)
    - Por eso no tiene sentido hablar de más de un argumento ni cero argumentos.
- El mecanismo de evaluación de las funciones es la **reducción**

1.3 Declaración de funciones

Las funciones son nuestras herramientas para poder operar valores. Para declarar una función, utilizamos el siguiente prototipo:

```
nombreFuncion :: tipoParametroUno -> tipoParametroDos -> ... -> tipoRetorno
nombreFuncion parametroUno parametroDos ... parametroN = función
```

- El operador igual (**=**) dice que dos expresiones *son equivalentes* (no se asigna nada).
- **nombreFuncion** no es más que un alias/etiqueta para la función
- Los **parámetros** pueden ser tantos como se necesite y se separan por espacios
- La **función** es lo que se debe realizar (y puede devolver cualquier tipo de valor)
    - Puede ser **infija**, es decir, va *en medio* “5 + 3”
    - O puede ser **prefija**, es decir, va *antes* “(+) 5 3”

*Las variables y las funciones comienzan con **minúscula**, mientras que los tipos con **mayúscula**.*

2. Inmutabilidad, Efecto y Transparencia Referencial

En Haskell no se produce **efecto**. Esto quiere decir que los valores igualados no van a mutar luego de ser operados por las funciones, es decir, son **inmutables**.

Cuando declaro una función o variable con un valor, el mismo no se cambiará nunca. Por eso, en Haskell tenemos **transparencia referencial.**

Si definimos lo siguiente en nuestro código:

```
nombre :: String
nombre = "Nombre"

agregarPunto :: String -> String
agregarPunto = (++ ".")
```

Y luego en GHCi jugamos con la función y la etiqueta **nombre**, veremos lo siguiente:

```
> nombre
"Nombre"

> agregarPunto nombre
"Nombre."

> nombre
"Nombre"
```

Como podemos ver **nombre** es un string ya definido en nuestro código (“Nombre”) que si le aplicamos la función **agregarPunto**, nos devuelve un *nuevo string* con el punto pero nunca modifica el original.

3. Pattern Matching

Consiste en definir una función en múltiples líneas, es decir, declaro sentencias que no contengan variables si no **patrones** (valores que puede tomar un parámetro).

```
cancionFavoritaDe :: String -> String
cancionFavoritaDe "Mauro" = "Cancion X"
cancionFavoritaDe "Elisa" = "Cancion Y"
cancionFavoritaDe alguien = "Cancion Desconocida" - (1)

- También podriamos utilizar la variable anónima “_” donde (1) = (2)
cancionFavoritaDe _ = "Cancion Desconocida" - (2)
```

Como podemos ver a los nombres “Mauro” y “Elisa” les asociamos una canción en particular, y para el resto de casos es *desconocida*.

> **OBSERVACIÓN:** El programa siempre toma la primera opción que cumpla la condición por eso primero siempre se ponen los patrones.

Usar **pattern matching** es crucial para contemplar los casos especiales de un problema, sin embargo, si quisiéramos acceder a los elementos de tuplas, estaríamos fuertemente arraigados al orden de los elementos (no soporta cambios en las estructuras)

4. Guardas

Las funciones *por partes* son aquellas que para diferentes valores del dominio, tienen una definición diferente. En este caso, estos *distintos dominios* se representan a través de **guardas** (**|**)

```
modulo :: Number -> Number
modulo numero
  | numero >= 0 = numero
  | otherwise = -numero – otherwise se utiliza para todo el resto
```

Como podemos ver, la función **modulo** recibe un número que, en caso de ser positivo o cero devuelve el mismo, y en caso de ser negativo devuelve el número pero negativo.

Internamente, una guarda tiene este prototipo (donde la expresión booleana puede ser compuesta)

```
| expresionBooleana = retorno
```

OBSERVACIÓN: Al igual que en el *pattern matching* el programa siempre toma la primera opción que cumpla la condición.

4.1 Guarda con las guardas

No se debe utilizar guardas en expresiones donde el valor de verdad se pueda determinar programáticamente

```
esMayorA :: Number -> Number -> Bool
esMayorA num1 num2
  | num1 > num2 = true
  | num1 < num2 = false
```

Es una aplicación **incorrecta** de guardas ya que la función **esMayorA** se debería declarar así:

```
esMayorA :: Number -> Number -> Bool
esMayorA num1 num2 = num1 > num2
```

Donde la expresión booleana “num1 > num2” ya de por sí resuelve la operación

Otro error común es olvidar el *otherwise* o *repetir lógica*.

5. Tipado

Es importante que las funciones tengan un **tipado** bien definido ya que nos permite restringir el dominio de la función y con ello, tener un mayor control de la imagen.

5.1 Tipado de Funciones

```
nombreFuncion :: tipoParametroUno -> tipoParametroDos -> ... -> tipoRetorno
nombreFuncion parametroUno parametroDos ... parametroN = operacion
```

Como previamente habíamos mencionado en la primera sección, el tipado es parte del prototipo de la función y como tal, define el tipo de los parámetros y el del valor del retorno.

Haskell está preparado para **inferir** estos tipados, es decir, es capaz de reconocer los tipos de datos de los parámetros y ajustarlos a la función. Si bien Haskell se encargará de hacerlo lo más **genérico** posible (para que no tengamos problemas) esto puede ser un problema, por lo que en ciertos escenarios, es necesario restringir el dominio a través de desarrollar el tipado de la función.

5.2 Variables de Tipo

En Haskell, las **variables de tipo** son una característica poderosa que permite escribir código genérico y polimórfico.

Permiten definir funciones y tipos de datos que funcionan con diferentes tipos sin especificar explícitamente cuáles son esos tipos. Esto promueve la reutilización de código y aumenta la flexibilidad del sistema de tipos en Haskell.

Se representan a través de letras minúsculas como a, b, c.

```
identidad :: a -> a
identidad parametro = parametro
```

```
ignorarPrimerParametro :: a -> b -> b
ignorarPrimerParametro parametroUno parametroDos = parametroDos
```

Ambas son funciones que admiten cualquier tipo de dato.

5.3 Typeclasses

Las **typeclasses** definen una lista de funciones que un tipo debe implementar para considerarse de esa clase. No es un tipo, es una restricción sobre una *variable de tipo* (que puede tomar como valores posibles tipos concretos).

```
sonIguales :: Eq a => a -> a -> Bool
sonIguales parametroUno parametroDos = parametroUno == parametroDos
```

Como podemos ver, anteponemos **Eq a =>** para indicar que *"a"* tiene que ser una variable de tipo equiparable (de otra forma, no podríamos evaluar "parametroUno == parametroDos").

Luego, también tenemos **Ord** (para decir que sean ordenables) y **Show** (para decir que se puedan mostrar por consola *GHCi*).

6. Data

Utilizamos el **data** para definir un nuevo tipo de dato (en la programación estructurada sería el equivalente a un *struct*). El prototipo es el siguiente:

```
data NombreTipo = Constructor{
  primerCampo :: Number,
  segundoCampo :: String,
  tercerCampo :: [String]
}deriving(Show,Eq,Ord)
```

Como podemos ver, se define un nombre para el tipo de dato <**NombreTipo**> seguido de un constructor <**Constructor**> y luego, entre llaves, los campos del tipo de dato (seguidos de su tipo).

Al final, podemos elegir qué typeclasses heredará nuestro data a través de *deriving()*.

6.1 Instanciación

Luego, podemos instanciar una "*variable*" de este tipo de dato nuevo a través de las siguientes formas:

```
algoDelTipo :: NombreTipo
algoDelTipo = Constructor {primerCampo = 1, segundoCampo = "Hola",
tercerCampo = ["Hola","Mundo"]}

otraCosa:: NombreTipo
otraCosa = Constructor 2 "Chau" ["Chau","Mundo"]
```

Siendo que la primera forma de declarar el tipo es mejor ya que se abstrae del orden del *data*, en cambio, en el segundo caso, si nosotros decidimos cambiar el orden de los campos del *data*, tendremos efectos no esperados.

6.2 Funciones de acceso

Luego, también existen las **funciones de acceso** que toman al constructor y los tipos de los campos entre paréntesis y devuelven el campo que queremos (es decir, un *pattern matching*).

```
> tercerCampo algoDelTipo
[ "Hola", "Mundo"]
```

Siguiendo con el ejemplo anterior, la función para acceder al tercer campo, no es más que el nombre del mismo campo, es decir “tercerCampo”.

6.2.1 Definir tus propias funciones de acceso

Se puede acceder a un campo del data de la siguiente forma

```
hola (Constructor _ _ tercerCampo) = tercerCampo
```

Luego en GHCi obtendremos estos resultados:

```
> hola algoDelTipo
[ "Hola", "Mundo"]
> tercerCampo algoDelTipo
[ "Hola", "Mundo"]
```

Donde podemos apreciar que a efectos prácticos **hola** es lo mismo que **tercerCampo**, ambas funciones de acceso al tercer campo del data *NombreTipo*

6.3 Type Alias

Los **type alias** o simplemente **type**, son otra forma de llamar a los tipos. Generalmente se utilizan para dar mayor expresividad al contexto en el que se está programando (pero nunca construye otro tipo).

```
type Lista = [String]
```

Podríamos definir este nuevo type *Lista* y en nuestra estructura, reemplazar el tipo en el tercer campo:

```
data NombreTipo = Constructor{
  primerCampo :: Number,
  segundoCampo :: String,
  tercerCampo :: Lista
}deriving(Show,Eq,Ord)
```

Y todo seguiría funcionando de la forma esperada (en *c* es lo mismo que utilizar un *typedef*)

En ciertos casos, está bueno utilizarlo ya que abstraen tipos de datos más complejos como:

```
type Algo = String -> String -> [String]

unaFuncionDeStrings :: String -> String -> [String]

otraFuncion :: Algo -> [String]
-- Sería lo mismo que String -> String -> [String] -> [String]
```

7. Tuplas

Es un *data anónimo*, no tiene nombre. Se ponen entre paréntesis los datos que quieras (separados por comas). Las tuplas tienen una *aridad*, que es la cantidad de datos que tiene.

Cada parámetro tiene un tipo, es por ello por lo que el tipo de la tupla son los tipos de los parámetros, entre paréntesis y separados por comas:

```
type Alumno = ("Mauro","289-227.1",21)
```

```
- El tipado de la tupla es:
t: (String,String,Number)
```

Las **tuplas** se utilizan para modelar datos muy genéricos, porque para tuplas con el mismo tipo de datos, no se puede diferir en su contexto. Es decir, con type Alumno también podríamos definir:

```
type Alumno = ("Coca Cola","Estados Unidos",3000)
```

Siendo que en esencia *Coca Cola* no es un alumno (pero sí aplica a la tupla).

Para acceder a un elemento de la tupla se puede hacer de la siguiente forma:

```
legajo (_,unLegajo,_) = legajo
```

8. Aplicación

Cuando se aplica una función a unos argumentos, el resultado se obtiene sustituyendo esos argumentos en el cuerpo de la función respetando el nombre que se les dio a cada uno.

El resultado producido puede no ser reducido (es decir, ser el resultado final) o bien, que dicho resultado sea otra expresión que contenga otra aplicación de funciones.

8.1 Aplicación Parcial

Hablamos de **aplicación parcial** cuando una función recibe menos parámetros de los que espera, por lo que retorna *una función*.

Todas las funciones de Haskell pueden ser aplicadas parcialmente ya que están currificadas (se guardan los parámetros pasados hasta recibir los que le faltan).

```
sumarUno:: Number -> Number
sumarUno numero = numero + 1

sumarUno:: Number -> Number
sumarUno = (+ 1)
```

Como podemos ver, ambas funciones hacen lo mismo, con la diferencia que la segunda versión, toma la función suma (+) y recibe un solo parámetro y espera otro *Number*.

La aplicación parcial siempre sucede "después" del igual (=)

8.1.1 Notación Point-Free

Puedo eliminar la aplicación de un parámetro cuando se usa de un lado y del otro, a la derecha de todo. No cambia el tipo de la función ni su funcionamiento. No es posible en todas las funciones.

```
obtenerNotaFunc :: Alumno -> Number
obtenerNotaFunc al = valor . notaFuncional $ al

obtenerNotaFunc :: Alumno -> Number
obtenerNotaFunc = valor . notaFuncional
```

Como podemos ver, la segunda versión no utiliza el alumno *"al"* pues se *cancela* en ambos miembros.

9. Composición

Lo mismo que con la composición de funciones matemáticas $f\ o\ g\ (x)$, primero se aplica la función de la derecha con el valor y luego se aplica la de la izquierda con el valor devuelto de $g\ (x)$ quedando como: $f\ (g(x))$.

Al igual que en matemática, la *Imagen* de la función *g* tiene que ser coincidente con el dominio de la función *f* para que se pueda operar.

Por eso, la composición siempre es **hacia la derecha**.

9.1 Operador de composición

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

La función de composición (**.**) toma dos funciones como argumentos y devuelve una nueva función. Aquí se desglosa el tipo:

- (b -> c): Es el tipo de la primera función, que toma un argumento de tipo b y devuelve un valor de tipo c.
- (a -> b): Es el tipo de la segunda función, que toma un argumento de tipo a y devuelve un valor de tipo b.
- a -> c: Es el tipo de la nueva función resultante de la composición, que toma un argumento de tipo a y devuelve un valor de tipo c.

Si quisiéramos componerlo al revés hacia la izquierda no podríamos ya que muy probablemente, tengamos un problema de tipado (a no ser que toda la función sea de un mismo tipo).

Por esta misma razón necesitamos utilizar el (**.**), sino, estaríamos pasando una función como parámetro (dependiendo del contexto, puede ser lo *no esperado*)

Lo que nos queda después de componer dos funciones es una ***nueva función***

Como no podemos pisar valores con variables, la composición nos permite encadenar las funciones para trabajar con diferentes valores y así poder crear soluciones más complejas.

```
cuadruple :: Number -> Number
cuadruple n = doble (doble n) - Sin composición

cuadruple :: Number -> Number
cuadruple = doble . doble - Con composición y aplicación parcial
```

9.2 Composición de muchos parámetros

Usamos la **aplicación parcial** sobre funciones que esperan n parámetros para currificarlas. Pongo los parámetros que *están a la izquierda* (es decir, los primeros parámetros) para que reciba otro.

Si me interesa recibir el segundo parámetro (y no el primero) podemos utilizar la función **flip**, cuyo tipado es:

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

Y se utiliza de la siguiente forma:

```
concatenarStrings :: String -> String -> String
concatenarStrings s1 s2 = s1 ++ s2

invertirStrings :: String -> String -> String
invertirStrings = flip concatenarStrings
```

Que en GHCi se vería algo como esto:

```
> invertirStrings "la" "ho"
"hola"
```

10. Orden Superior

Una función es de **orden superior** si recibe otras funciones por parámetro o devuelven una función como resultado.

Las funciones más comunes de orden superior son: *filter, map, fold, funciones de ordenamiento o búsqueda, (.) composición, ($) la aplicación, flip*, etc.

```
aplicarOperacionNum :: (Number -> Number) -> Number -> Number
aplicarOperacionNum funcion número = funcion número

aplicarSumarUno :: Number -> Number
aplicarSumarUno = aplicarOperacionNum sumarUno
```

Como podemos ver, la función **aplicarOperacionNum** es de orden superior ya que recibe una función que va de *Number a* Number y su tipado se pone entre paréntesis.

En caso de **aplicarSumarUno** sí le pasamos una función por parámetro, no utilizamos composición.

11. Expresiones Lambda

Las **expresiones lambda** son una herramienta que sirve para crear funciones *descartables* (que no tienen ni nombre).

Son utilizadas cuando se necesita delegar una función a sabiendas de que no será reutilizada (y por ende, no tiene sentido crearla). Se define la función en el momento y se la pasa como un parámetro.

Se aclara los parámetros que recibe y lo que retorna a través del siguiente prototipo:

```
(\ <parámetros> -> <expresión>)
```

Un ejemplo de uso para las expresiones lambda es este, junto con el uso de **map** (ord. superior):

```
triplicarLista :: [Number] -> [Number]
triplicarLista lista = map (\x -> x * 3) lista
```

Donde map aplicará la función “x*3” a todos los elementos de la lista.

```
> triplicarLista [1,2,3,4]
[3, 6, 9, 12]
```

Si no le damos un nombre a nuestras funciones, podríamos perder posibles **abstracciones** útiles en otro momento, es importante saber cuándo utilizar una expresión lambda.

12. Listas

Siempre la lista es de un tipo y **NO** se pueden mezclar distintos tipos en las listas

La lista [a] tiene dos constructores:

- [ ] : Que representa una lista vacía (es el *caso semilla*)
- (:) : Que separa el primer elemento **head** (o cabeza) de la lista restante, **tail** (o cola)

Las **listas** son estructuras recursivas (compuestas por una cabeza y una cola, *osea lista*). Si la lista no está vacía, podemos dividirla en cabeza y cola hasta que esté vacía

```
[1,2,3,4] == 1:2:3:4:[]
1:[2,3,4] == 1:2:[3,4] == 1:2:3:[4] == 1:2:3:4:[]
```

12.1 Funciones de listas en bajo nivel

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

tail :: [a] -> [a]
tail (x : xs) = xs

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

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

elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y : ys) = y ==x || elem x ys

estaOrdenada :: Ord a => [a] -> Bool
estaOrdenada [] =True
estaOrdenada [x] = True
estaOrdenada (x:y:z) = x<y && estaOrdenada z
```

12.2 Funciones de listas en alto nivel

Con estas funciones, producimos resultados a partir de cada elemento de la lista.

Antes, consideremos la siguiente lista con la cual explicaremos como se usa cada función:

```
lista :: [Int]
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

12.2.1 Map

La función **map** recibe una función y una lista, para aplicar la función a cada elemento de la lista, devolviendo esa nueva lista con todos los elementos aplicados. Su tipado es:

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

Donde si la aplicamos con la función anterior **triplicarLista** (que utilizaba map) tenemos este resultado:

```
> triplicarLista [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[ 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

12.2.2 Filter

La función **filter** recibe una condición y una lista, para luego devolver una lista filtrada solo con los elementos que cumplen dicha condición. Su prototipo es:

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

Filter podría ser utilizada de la siguiente manera:

```
pares :: [Number]
pares = filter even
```

Donde obtendríamos el siguiente resultado:

```
> pares [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 6, 8, 10]
```

12.2.3 All y Any

Ambas funciones reciben como parámetro una condición (que es una función) y una lista, ambas retornando un booleano.

La función **all** nos dice si TODOS los elementos de la lista cumplen la condición. Su tipado es:

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

Un ejemplo de uso sería:

```
todosMenoresQue20 :: Bool
todosMenoresQue20 = all (< 20) lista
```

```
> todosMenoresQue20 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
True
```

La función **any** nos dice si ALGUNO de los elementos de la lista cumple la condición. Su tipado es:

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

Un ejemplo de uso sería:

```
algunoMayorQue8 :: Bool
algunoMayorQue8 = any (> 8) lista
```

```
> algunoMayorQue8 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
True
```

También podría usar filter y ver si el tamaño de la lista final coincide con el de la lista inicial, o si el tamaño de la lista final es igual a 0. Pero no tiene sentido. Es entonces que…

> **ADVERTENCIA**: Siempre que tenga un problema busco la herramienta que esté más cerca de ese problema para resolverlo. No hay que distraerse buscando algoritmos si tengo herramientas que me lo resuelven. Por eso, consultar la **guía de lenguajes**

12.3 Loop infinito

Se dice que hay un **loop infinito** cuando una función no termina nunca, ya que no hay ningún punto en el que se corte la recursividad.

El manejo de la infinitud puede ser combatido a través de la evaluación perezosa.

12.4 Listas por compresión

Son un azúcar sintáctico que nos permite armar listas a partir de los elementos de otra luego de aplicar filtros y transformaciones.

- Listas por Rangos: **[1 … 100]**
- Listas Infinitas: **[1..]**

12.5 Fold

A partir de un conjunto, me interesa llegar a un resultado (es decir, *currificación*)

La función **fold** recibe como parámetros una función reductora, una semilla y una lista para que nos retorne algo.

Su prototipo es:

```
fold :: (a -> b -> b) -> b -> [a] -> b
```

La semilla es un resultado para el caso de que b esté vacío. La función reductora agarra un resultado parcial y un elemento y calcular el resultado (parcial) con ese elemento incluido.

Cada elemento me genera un nuevo resultado parcial. Cada resultado parcial es la semilla para aplicar de nuevo la función con el siguiente elemento. Cuando "se me terminan" los elementos, obtengo el resultado final.

Ejemplo: fold (+) 0 [1, 2, 3] = 6

12.5.1 Tipos de fold

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

→ foldr1 :: (a -> a -> a) -> [a] -> a : La semilla es la head

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

→ foldr1 :: (a -> a -> a) -> [a] -> a : La semilla es la head
```

13. Definiciones Locales

Se pueden definir funciones que sólo se encuentran definidas en el contexto de una función a través de ***where***:

```
imc peso altura
 | peso / altura ^ 2 <= 18.5 = "Bajo Peso"
 | peso / altura ^ 2 <= 25.0 = "Peso Esperado"
 | otherwise = "Alto Peso"

imc peso altura
 | indiceGordura <= 18.5 = "Bajo Peso"
 | indiceGordura <= 25.0 = "Peso Esperado"
 | otherwise = "Peso Esperado"
 where indiceGordura = peso / altura ^ 2
```

OBSERVACIÓN: Al igual que con las guardas se debe dejar un espacio de indentación

Conceptos

Transparencia Referencial

Una función es **transparente** si es *determinista* (siempre da el mismo resultado para los mismos parámetros), independiente y no produce efectos secundarios.

```
doble :: Number -> Number
doble x = x * 2
```

En este caso la función **doble** es pura ya que su resultado únicamente depende del parámetro (por ejemplo, siempre que x valga 1, el doble será 2).

Currificación

Es una técnica que consiste en transformar una función que recibe múltiples argumentos en una secuencia de funciones que utilizan un **único** argumento.

Por ejemplo, **f :: a -> b -> c -> d**, es una función que recibe tres argumentos (a,b,c) y devuelve *d*.

La currificación en este caso sería tener una función: **f :: a -> (b -> c -> d)** es decir, que recibe *a* y devuelve otra función, que a su vez esa función sería: **f :: b -> (c -> d)** y luego devuelve otra función con el mismo criterio: **f :: c -> d**, donde se recibe finalmente, el valor *d*, y no una función.

Efecto

En el paradigma funcional no se produce **efecto**. Se recibe un parámetro y se retorna *emulando* como si se aplicara la modificación, aunque técnicamente, sólo se construye una estructura igual con las modificaciones aplicadas.

```
subirNota unaNota = Nota {valor = valor unaNota + 1}
```