Paradigma Funcional - Clase 1

Paradigma Funcional

Recordando un poco matemática, una **función** *f* es una relación entre un conjunto dado Dom_f(el dominio) y otro conjunto de elementos $Codom_f$ (el codominio) de forma que a cada elemento *x* del dominio le corresponde un único elemento del codominio *f(x)*. Se denota por:

$f :: Dom_f \rightarrow Codom_f$

Cumpliendo:

- Existencia: Todos los elementos de Dom_f están relacionados con elementos de $Codom_f$
- Unicidad: Cada elemento de Dom_f esta relacionado con un único elemento de $Codom_f$

Si *x* es un elemento del dominio, al elemento del codominio asignado por la función y denotado por *f(x)* se le llama **imagen** de la función *f* de *x*.

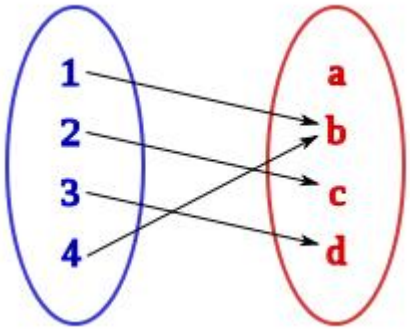

Figura 1

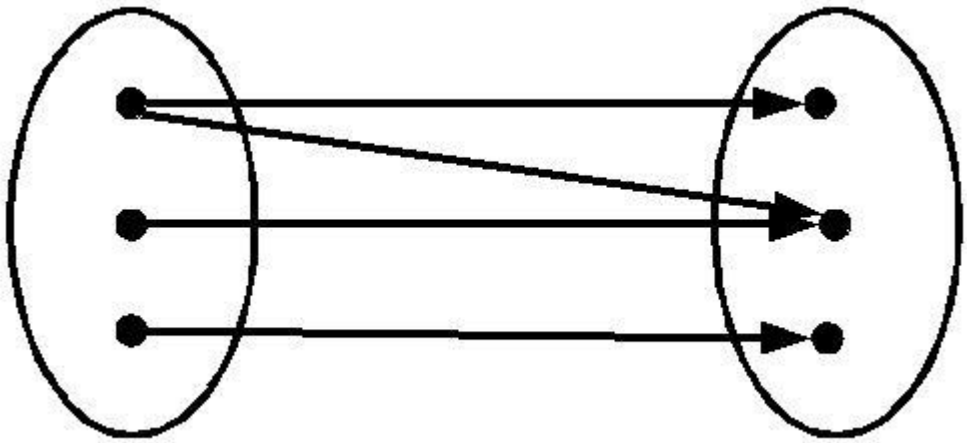

Figura 2

La **Figura 1** representa una función; la **Figura 2** no es función porque para un mismo elemento del dominio hay más de una imagen, eso se va a poder dar en lógico (múltiples respuestas) pero no en funcional .

El lenguaje que vamos a usar para poner en práctica los conceptos del paradigma funcional es Haskell.

Tomando como ejemplo la siguiente función:

f(x) = x + x

Para definirla en Haskell hay que indicar su **nombre**, un **nombre para cada uno de sus argumentos** y un **cuerpo** que especifica cómo se debe calcular el resultado en base a los argumentos.

```
doble x = x + x
----  -   -----
nombre args  cuerpo
```

Dentro de este lenguaje el signo = tiene el mismo significado de la definición matemática.
Es distinto del = de una ecuación, este es de una definición y se lee de izquierda a derecha.

El paradigma funcional está basado en conceptos que vienen de la matemática, entonces algunas cosas (p.ej. notaciones) están sacadas de lo que aprendimos en Análisis I / Álgebra / Discreta.

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.
Es posible que esto produzca un resultado que no puede ser *simplificado* (e.g. un número); pero es más común que el resultado de esa sustitución sea otra expresión que contenga otra aplicación de funciones, entonces se vuelve a hacer el mismo proceso hasta producir un resultado que no pueda ser *simplificado* (un resultado final).

Por ej., el resultado de aplicar `doble 3` (o sea, la función doble aplicada al número 3) se puede obtener de la siguiente manera:

```
doble 3
= { aplicamos la función doble }
3 + 3
= { aplicamos la función + }
6
```

Si queremos obtener el resultado de `doble (doble 2)` en donde la función doble es aplicada 2 veces, podemos hacer:

```
doble (doble 2)
= { aplicamos la función doble que está dentro del paréntesis}
doble (2 + 2)
= { aplicamos la función + }
doble 4
= { aplicamos la función doble }
4 + 4
= { aplicamos la función + }
8
```

También podemos obtener el mismo resultado aplicando primero la función doble de más afuera:

```
doble (doble 2)
= { aplicamos la función doble que está afuera del paréntesis }
doble 2 + doble 2
= { aplicamos la primer función doble }
(2 + 2) + doble 2
= { aplicamos la primer función + }
4 + doble 2
= { aplicamos la función doble }
4 + (2 + 2)
= { aplicamos la segunda función + }
4 + 4
= { aplicamos la función + }
8
```

El orden en que realicemos las **reducciones** (basta de decirle simplificar) no afecta al resultado final pero si a la eficiencia. Lo copado es que hay un motor que se encarga de esto (se llama **motor de reducciones** CUAC!)

Analicemos esto con el ejemplo `sum [1..5]`

```
sum [1 . . 5]
= { aplicamos [ . . ] }
sum [1, 2, 3, 4, 5]
= { aplicamos sum }
1 + 2 + 3 + 4 + 5
= { aplicamos + }
15
```

Al trabajar con Haskell tenemos por un lado a nuestro programa y por el otro un entorno de consultas.

En Haskell es muy importante la noción de **tipo**, con lo cual van a ver que el compilador los va a putear bastante si no los respetan.

Tipos

Ahora bien, tanto en la definicion matemática y como en la definicion en haskell están los conjuntos de partida (dominio) y de llegada (codominio).

En matemática si tuvieramos una definición de `doble` los conjuntos serían Z → Z (de enteros a enteros).
En Haskell es Integer → Integer

Esto quiere decir que desde Haskell vemos a los conjuntos que son dominio y codominio de la función como **tipos**; los tipos en funcional son los posibles dominios y codominios de las funciones.
Entonces por ahora ¿de qué tipo es 8? Es de tipo Integer (ya veremos las sutilezas de los distintos sabores de número, por ahora suponemos todos Integer).

Vamos a llamar "**expresión**" a cualquier cosa que puedo escribir en Haskell.nsdkgsdkg

```
expresión :: Tipo
```

El `::` se debe leer como ***"<lo que está a la izquierda> es de tipo <lo que está a la derecha>"***

Toda **expresión** tiene un tipo.
P.ej. doble es una función y también tiene un tipo. ¿cuál es?

En haskell podemos preguntar eso poniendo `:type expresionDeLaQueQuieroSaberElTipo`

```
>:t doble
doble :: Integer → Integer
```

... que es lo que dijimos en la definición. "función que va de enteros a enteros" también es un tipo, y `doble` es de ese tipo.

Otra expresión posible es `doble 4`, así como está, sin calcular. ¿De qué tipo es ese valor? Es Integer. ¿Cómo me doy cuenta? Porque doble va de Integer en Integer, entonces si le paso un Integer como parámetro, seguro que tiene que devolver un Integer. Para darme cuenta de esto razoné con tipos, no me hizo falta hacer ninguna cuenta. Efectivamente puedo preguntar

```
>:t doble 4
doble 4 :: Integer
```

Sí pasa que como una función (p.ej. `doble`) es un valor que tiene un tipo, puedo tratarla como cualquier otro valor; con una función puedo hacer básicamente cualquier cosa que puedo hacer con cualquier otro valor, p.ej. un entero o una lista (esto lo vamos a retomar la clase que viene).

Entonces, si programo definiendo funciones, los valores/datos/individuos que voy a manejar pertenecen a dominios y codominos de las funciones. Y puedo pensar para el tipo de cada función, qué valores tiene sentido que reciba y que devuelva.
Hasta ahora estamos tratando con valores sencillos, números y booleanos. Si quiero una función que me diga si un alumno puede cursar una materia, necesito un valor más complejo, para eso vamos a manejar estructuras más o menos parecidas a las que vimos en lógico (p.ej. listas).
Ahora sigamos aprendiendo características del paradigma funcional con valores sencillos, y después le metemos valores compuestos.

Aclaración importante: no es necesario definir el tipo de mi función en Haskell, dado que el motor sabe inferir a estos a partir de la implementacion que nosotros hagamos (como hicimos al principio).

Tipos básicos

Bool, Char, String, Int, Integer, Float, (A → B) *Funciones* - hay más pero con esto alcanza por ahora

Todo esto nos da a entender que haskell es un lenguaje que se fija mucho en los tipos de las cosas que maneja, en esto es distinto a Prolog.

Esto ¿quiere decir que el paradigma funcional siempre se fija en los tipos?

Noooooo eso va con el lenguaje, no con el paradigma. Hay lenguajes funcionales relajados respecto de los tipos (Lisp) e implementaciones en las que sí se declaran y

chequean tipos (Haskell, Scala, etc.). Cerca del final de la cursada vamos a hablar de esto.

Boludeando con Haskell

```
> 2 + 3
5
> 2 − 3
−1
> 2 * 3
6
> 7 `div` 2
3
> 2 ^ 3
8
```

Fíjense que el operador de división entera está escrito `div`.
Las funciones que reciben 2 valores pueden usarse
de forma **infija** (*valor1* funcion *valor2*) o de forma **prefija** (funcion *valor1* *valor2*).

Los **chirimbolos** (operaciones como +,-,*,/) que representan funciones de 2 parámetros se usan directamente de **forma infija**, para usarlo de **forma prefija** los ponemos entre **paréntesis**

```
> (*) 2 4
8
```

Las funciones que **no son chirimbolos** se usan directamente de **forma prefija**, si queremos usarlas de **forma infija** las encerramos entre **tildes graves (comillas simples a medio acostar a la izq)**.

```
> 4 `mod` 2
0
```

Siguiendo las convenciones matemáticas la exponenciación tiene una prioridad más alta que la multiplicación y la división, y estás últimas tienen mayor prioridad que la suma y la resta.
Por ej., 2 * 3 ^ 4 significa 2 * (3 ^ 4), y 2 + 3 * 4 significa 2 + (3 * 4).
Además, la exponenciación es asociativa a derecha, en cambio los otros operadores aritméticos asociacian a izquierda.
Por ej. 2 ^ 3 ^ 4 significa 2 ^ (3 ^ 4), mientras 2 − 3 + 4 significa (2 − 3) + 4.

Sin embargo, en la práctica es común usar paréntesis para evitar confusiones.

Aplicación de funciones

Notacion matemática
    f(a, b) + c d
En Haskell
    f a b + c * d

La aplicación se denota solo poniendo espacio entre la función y sus argumentos.
Además la aplicación de funciones que no son operadores tiene una precedencia mayor que la de los operadores (un **operador** es una función que recibe 2 parámetros y se usa de forma infija ya sea un chirimbolo o una función con ``)

Matemática	Haskell
f(x)	f x
f(x , y)	f x y
f(g(x))	f (g x)
f(x , g(y))	f x (g y)
f(x) g(y)	f x * g y

```
promedio ns = (sum ns) `div` (length ns)
```

Es equivalente a esto

```
promedio ns = sum ns `div` length ns
```

O bien,

```
promedio ns = div (sum ns) (length ns)
```

Conjuntos en Haskell: Listas

Dijimos que las funciones por cumplir con la propiedad de unicidad solo pueden tener un elemento en el codominio para un elemento del dominio dado.

A pesar de esto, existen funciones que intuitivamente parece que retornan más de un valor.

Por ejemplo, una función `notas` que recibe un alumno y devuelve las notas que obtuvo en los parciales de este cuatrimestre (asumamos que son las notas de todas sus materias, eso no es relevante)

Si la función **debe** cumplir con la propiedad de unicidad su definición de dominio e imagen sería:

```
notas :: Alumno -> ???
```

Pero cuál es el tipo de la imagen `???` ?

En vez de pensar las notas de un alumno dado como un montón de valores podemos verlo **como un solo valor**, un valor compuesto.

En una primera aproximación podemos decir entonces

```
notas :: Alumno -> Conjunto de Números
```

En Haskell vamos a llamar a los conjuntos **listas**.

Una lista es un valor compuesto

- **los elementos que forman la lista deben ser del mismo tipo**
- **la lista no tiene una cantidad de elementos determinada**

-- Una lista de números

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

-- Otra lista de números

```
[1325,12,512,512,5,12,312,3]
```

-- Una lista de caracteres
```
['h','o','l','a'] = "hola"
```

-- Una lista de booleanos
```
[True,False,False]
```

En Haskell no es posible tener una lista con elementos que no sean del mismo tipo

-- Una lista que no es posible tener en Haskell
~~[1,2,3,True]~~

El prelude (la bibilioteca estandar) trae un montón de funciones útiles sobre listas. En Haskell, los elementos de la lista se encierran por corchetes y se separan por comas.

• head/1
Recibe una lista y devuelve el primer elemento de la misma
```
> head [1, 2, 3, 4, 5]
1
```
• tail/1
Recibe una lista y devuelve una lista con todos los elementos excepto el primero (recibe una lista y devuelve la cola)
```
> tail [1, 2, 3, 4, 5]
[2, 3, 4, 5]
```
• (!!)/2
Recibe una lista y un número que representa una posición y devuelve el elemento de la lista que está en esa posición (base 0)
```
> [1, 2, 3, 4, 5] !! 2
3
```
• take/2
Recibe un número que representa una cantidad de elementos y una lista y devuelve una lista con los primeros elementos hasta llegar a la cantidad
```
> take 3 [1, 2, 3, 4, 5]
[1, 2, 3]
```
• drop/2
Recibe un número que representa una cantidad de elementos y una lista y devuelve lo que no devuelve take
```
> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```
• sum/1
Recibe una lista de números y devuelve su sumatoria (o sea la suma de todos sus elementos)
```
> sum [1, 2, 3, 4, 5]
15
```

• length/1
Recibe una lista y devuelve la cantidad elementos que tiene

```
> length [1, 2, 3, 4, 5]
5
```

• product/1
Recibe una lista de números y devuelve su productoria (o sea la multiplicación de todos sus elementos)

```
> product [1, 2, 3, 4, 5]
120
```

• (++)/2
Recibe 2 listas y devuelve una lista con los elementos de la primer lista y los elementos de la segunda (en ese orden)

```
> [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]
```

• reverse/1
Recibe una lista y devuelve una lista que tiene los mismos elementos pero en el orden inverso

```
> reverse [1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
```

Algunas funciones pueden dar error dados ciertos argumentos

```
> head []
Error
> 1 `div` 0
Error
```

Los Tipos y las Listas

Es importante separar el mundo de los tipos (eso que escribimos a la derecha del `::`) y el mundo de los valores

Ejemplo de listas de números enteros (valores)

```
-- Una lista con 4 números enteros
[3,2,1,8]

-- Una lista con 8 números enteros
[1325,12,512,512,5,12,312,3]

-- Una lista con 1 número entero
[5]

-- Una lista vacía (una lista sin elementos)
[ ]
```

Ahora bien, si escribimos

-- El tipo de la lista [3,2,1,8]

```
[3,2,1,8] :: Lista de Enteros
```

Lo que pasa es que Haskell tiene una forma particular de decir el tipo `"Lista de Enteros"`, esa forma es `[Integer]`

```
[3,2,1,8] :: [Integer]
[1325,12,512,512,5,12,312,3] :: [Integer]
[1 .. 100] :: [Integer]
[5] :: [Integer]
```

Podemos preguntarle a Haskell los tipos de otras listas

```
> :t [True,False,False]
[True,False,False] :: [Bool]

> :t "hola"
"hola" :: [ Char ]
```

Importante: no confundir las listas que solo tienen un elemento (e.g. [1] , "h" y [True]) con los tipos de las listas (e.g. [Integer], [Char] y [Bool])

Nota: `[ Char ] = String`

"Registros" en Haskell: Tuplas

Volviendo a nuestro ejemplo de la función notas, nos gustaría que recibiera un alumno como parámetro y nos retornara una lista de números con las notas de dicho alumno.

Podemos representar al "valor alumno" como un valor compuesto por

- un nombre (con un String)
- un apellido (con un String)
- un legajo (con un Integer)
- una lista de notas (con una lista de números)

Este valor compuesto cumple con las siguientes propiedades

- **Tiene un número fijo de elementos** (en este caso 4)
- **Los elementos que lo componen no son nesariamente del mismo tipo**

Por lo anterior, no nos sirve usar una lista tenemos que usar otro tipo de estructura, una especie de "registro". En Haskell a estos valores compuestos lo llamamos **tuplas**.

Para simplificar el ejemplo vamos a tratar a un alumno cono una tupla de 2 elementos, donde el primero es un String que representa su nombre completo y el segundo es su lista de notas.

Ya existen en el Prelude funciones para manipular tuplas de 2 elementos

```
-- Retorna la primer componente de la tupla
> fst (123, "hola")
123

-- Retorna la segunda componente de la tupla
> snd (123, "hola")
"hola"
```

Ahora podemos definir nuestra función notas de la siguiente forma

```
notas unAlumno = snd unAlumno
```

Algunas consultas

```
> notas ("Haskell Curry", [10,9,8])
[10,9,8]

> notas ("Edsger Dijkstra", [4,5,6])
[4,5,6]
```

Nota: para manejar tuplas de más de 2 elementos hay que esperar a la siguiente clase

Definiendo funciones

Ya vimos una forma de definir funciones, ahora vamos a ver otras

Expresiones lambda

Son funciones anónimas. Por ej., una función anónima que toma solo un número como argumento y produce el resultado x + x se puede definir como

```
  λx → x + x
```

En Haskell:

- λ se escribe \
- → se escribe ->

Esa definición es justamente la función doble que definimos antes, por eso podemos escribir

```
doble = λx → x + x
```

A pesar de no tener nombre, pueden usarse como cualquier otra función:

```
>(λx → x + x) 2
4
```

Una expresión lambda puede recibir más de 1 parámetro separándolos por espacio dentro de la expresión

```
--cuentaLoca es una función que recibe 3 parámetros
cuentaLoca = (\x y z -> x * 2 - y + 10 * z)
```

Definición con guardas

Se pueden ver como la forma que tiene Haskell de definir **"funciones partidas"**.

Si la primer guarda es True entonces se elige el primer resultado, si la segunda guarda es True se elige el segundo resultado, y así sucesivamente

La función *valor absoluto* se puede definir así:

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

El símbolo | se puede leer como "si es cierto que", la guarda otherwise está definida en el prelude como `otherwise = True`.

Haskell es un lenguaje tabulado!!! Importa que le pongan un "tab" delante de los |

Comentarios

La forma de poner comentarios en Haskell es

-- Comentario unilínea
{- Comentario multilínea -}