Paradigma Funcional - Clase 2

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 \ y → 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)
```

Aplicación parcial (ver más abajo Currificación)

Definamos la función `suma` (recibe 2 parámetros y devuelve su suma)

```
suma x y = x + y
```

Se puede pensar que suma recibe un solo número representado por **x** y devuelve una función que espera un solo numero **y.**
Esa idea expresada con expresiones lambda podría escribirse como

```
suma = λx → (λy → x + y)

> suma 2 3
5
```

Veamos como se reduce esto

```
(λx → (λy → x + y)) 2 3
= { aplicamos la expresión lambda que recibe x }
(λy → 2 + y) 3
= { aplicamos la expresión lambda que recibe y }
2 + 3
= { aplicamos la función + }
5
```

Aunque no usemos expresiones lambda para definir explícitamente que se recibe un parámetro por vez y que se devuelve una nueva función que espera el resto de los parámetros **todas las funciones (usando cualquier forma de definición) tienen esta propiedad.**

Imaginen que ⊕ representa un posible operador (una función que recibe 2 parámetros usada de forma infija).

Los distintos usos de un operador son los siguientes:

1. (⊕) ≈ λx → (λy → x ⊕ y)
   Una función que espera 2 parámetros
2. (x ⊕) ≈ λy → x ⊕ y
   Una función que espera 1 solo parámetro (el operando de la derecha)
3. (⊕ y) ≈ λx → x ⊕ y
   Una función que espera 1 solo parámetro (el operando de la izquierda)

Si el operador ⊕ es conmutativo las definiciones 2 y 3 son equivalentes (SOLO SI ES CONMUTATIVO, p.ej. ==, *, +, -, `max`, `min`)
Las funciones mod, div, /, etc. no son conmutativas

Tomando funciones concretas para ⊕

`(*2)` es la función doble, equivalente a definir una lambda así `λx → x * 2`
`(+)` es la función suma, equivalente a definir una lambda así `λx → (λy → x + y)`
`(1+)` es la función sucesor, equivalente a definir una lambda así `λy → 1 + y`
`(1/)` es la función recíproco o a la menos uno, equivalente a definir una lambda así `λy → 1 / y`
`(/1)` es la función dividir por uno, equivalente a definir una lambda así `λy → y / 1`
`(/2)` es la función mitad, equivalente a definir una lambda así `λx → x / 2`

Composición

El operador ◦ (en Haskell es simplemente un punto **.**) retorna la composición de 2 funciones como una única función, y puede definirse como

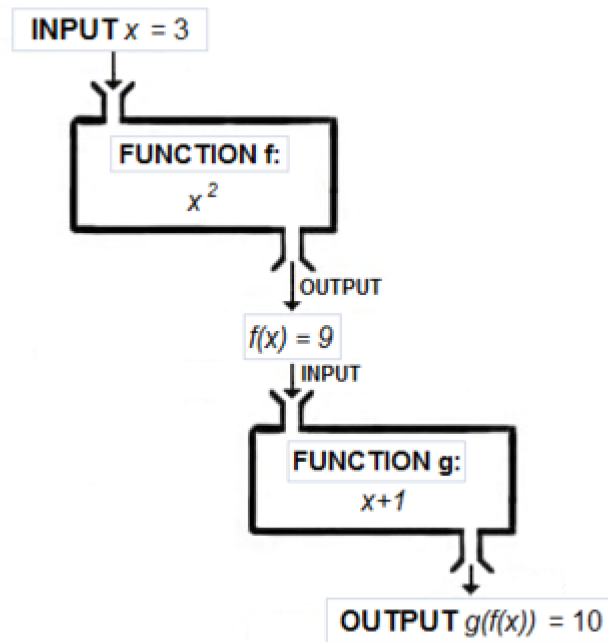
El dominio del primero tiene que estar incluido en la imagen del segundo; o sea, el dominio de g está incluido en la imagen de f
(◦) :: (b → c) → (a → b) → (a → c)

Primero se aplica **f** al parámetro **x** y después se aplica **g** a **f(x)**
g ◦ f = λx → g (f x)

Importante: la composición de funciones se lee de derecha a izquierda

(g ∘ f ∘ h)(x)

- x tiene que pertenecer al dominio de h
- h(x) le llega como parámetro a f
- f(h(x)) le llega como parámetro a g
- el resultado g(f(h(x))) pertenece a la imagen de g



Siguiendo el ejemplo del gráfico

```
cajitas input = (input ^ 2) + 1
```

```
> cajitas 3
10
```

Para verlo mejor podemos definir explícitamente a `f` y `g`

```
f x = x ^ 2
g x = x + 1
cajitas input = (f ∘ g) input
```

O bien,

```
cajitas ~~input~~ = (f ∘ g) ~~input~~
cajitas = (f ∘ g)
```

Usando expresiones lambda y sin definir aparte ni a `f` ni a `g`

```
cajitas input = ((λx → x +1) ∘ (λx → x^2)) input
```

Puedo "simplificar" el parámetro en ambos lados de la definición

```
cajitas = (λx → x +1) ∘ (λx → x^2)
```

Aprovechando la aplicación parcial y que la suma (+) y la potenciación (^) son operadores

```
cajitas = (+1) ∘ (^2)
```

Otro ejemplo:

```
odd n = not (even n)
odd = not ∘ even
```

Ejemplo a modo resumen

Hagamos un repaso de todo con la función `esPar`. La función `esPar` recibe un número (Integer) y devuelve un Booleano

```
esPar x = mod x 2 == 0
```

Si queremos definir `esPar` con una expresión lambda

```
esPar = λx → mod x 2 == 0
```

Si queremos definir `esPar` componiendo funciones

```
esPar = (esCero)∘(restoDivisionPorDos)

esCero y = 0 == y
restoDivisionPorDos x = mod x 2
```

Si queremos definir lo mismo sin usar funciones auxiliares

```
esPar = (0==)∘(λy → mod y 2)
```

En está última definición, la función == está aplicada parcialmente.

- A == que espera 2 parámetros le envío un solo parámetro y me devuelve una nueva función (0==) que espera un solo número y me devuelve un booleano.
- En `(λy → mod y 2)` usamos una expresión lambda para definir una función que recibe un solo número y me devuelve un número que representa el resto de su división por 2.
- Por último componemos esas 2 funciones `(0==)∘(λy → mod y 2)` para generar una nueva función que recibe un número y me devuelve un booleano (True si el resto de su división por 2 es igual a 0 y False en otro caso)

¿Por qué no funciona esto? `esPar = (0==)∘(mod 2)`

No funciona porque (mod 2) tiene la forma (x ⊕) espera el divisor, mod sin los `` no es un operador

¿Y esto? `` esPar = (0==)∘(`mod` 2) ``

Funciona porque (`mod` 2) tiene la forma (⊕ y) espera el dividendo

Currificación

Está idea surge como un acercamiento distinto al manejo de funciones con múltiples parámetros en donde en vez de tener una única función que recibe bocha de parámetros la transformamos en una cadena de funciones en donde cada una recibe un solo parámetro.

Para poder aplicar parcialmente una función, la función tiene que estar currificada. Se dice que una función está currificada si recibe sus parámetros de a uno por vez. Por cada parámeto que recibe devuelve una función que espera el resto de los parámetros. No se asusten por el término raro porque todas las funciones en Haskell están currificadas.

mult :: Int → (Int → (Int → Int))
mult x y z = x * y * z

Convenciones (gracias a ellas no tenemos que poner infinitos paréntesis):

- **La flecha del tipo de las funciones** asocia a derecha

```
Int → Int → Int → Int
```

es equivalente a

```
Int → (Int → (Int → Int))
```

- **La aplicación de funciones** (que la escribíamos simplemente con un espacio) se asume que asocia a izquierda

```
mult x y z
```

es equivalente a

```
( ( mult x ) y ) z
```

Las funciones que veían en Álgebra **no** están currificadas, por ejemplo $f(x,y) = x^2 + y^2$. f no recibe sus parámetros de a 1 por vez es como si tiene un solo parámetro que representa un punto en R^2

¿Qué vimos hasta ahora?

- La operación que vamos a realizar en el paradigma funcional es **aplicar funciones**
- Entonces, gran parte de la programación funcional va a consistir en definir funciones para luego poder aplicarlas
- Diferentes formas de definir funciones
    - Definiendo los argumentos y el cuerpo de la función

      ```
      -- Necesito una función que me diga si una nota está aprobada
      -- Recordamos el tema del dominio y la imagen, la función
      recibe un número y me devuelve un booleano

      esNotaAprobada :: Int → Bool
      esNotaAprobada nota = nota >= 4
      ```

    - Usando expresiones lambda para definir funciones sin nombre

      ```
      (λnota → nota >= 4) :: Int → Bool
      ```

    - Usando funciones aplicadas parcialmente (o sea, que reciben menos parámetros de los que esperan en total)

      ```
      (4<=) :: Int → Bool
      (>=4) :: Int → Bool
      ```

    - Nos queda pendiente repasar guardas y composición pero vamos a un ejemplo

Ejemplo

Imagínense que tenemos que representar a un **alumno**, y de cada alumno nos interesa su nombre y las notas que se saco en los 3 parciales de paradigmas.

A veces me va interesar ver **al alumno como un todo** compuesto por estos 2 valores y a veces me va a interesar **ver cada elemento que compone al alumno** por separado ...

Eso debería sonarles a una estructura de datos, una estructura de tamaño conocido ... en Pascal le decían registro acá le vamos a llamar **<u>tupla</u>**.

Si quiero hacer una función que me diga si un alumno aprobó ¿cuál sería su dominio y su imagen?

Estaría bueno que reciba un alumno y me devuelva un booleano

```
estaAprobado :: Alumno → Bool
estaAprobado unAlumno = ...
```

Y notas que es?? Podría ser una tupla de 3 elementos pero créanme que es copado que sea una lista

```
> estaAprobado ("Beto Velez",[4,8,9])
True
```

Decimos que un alumno aprobó si se sacó 4 o más en cada uno de sus parciales

```
estaAprobado unAlumno = (notas !! 0 >= 4) && (notas !! 1 >= 4) && (notas !!
2 >= 4)
    where notas = snd unAlumno
```

Ejemplo Reloaded

Todo muy lindo, pero sabés que ... vamos a empezar a tomar parcialitos. Por ahora son 2 parcialitos más 3 parciales y todos tienen que tener una nota mayor a o igual a 4 para aprobar, el tema es que los parcialitos son opcionales así que puede haber gente que tenga 3 notas, otra gente con 4 notas y otra gente con 5 notas (los parcialitos tienen sus beneficios la gente no es tonta).

Ahora tiene más sentido que hayamos representado a las notas de un alumno con una lista porque no tiene un tamaño fijo.

El problema es que es engorroso andar preguntando por cada elemento de la lista de notas cuando queremos chequear algo *"para todos los elementos"*

→ Orden

~~→ Orden~~

~~→ Orden~~

Bueno ... podríamos usar una función que haga el laburo pesado por mí.
Esa función puede recibir una lista de notas y una condición, devolvería True si todas las notas cumplen esa condición y False en caso contrario.

Esa función mágica en Haskell se llama `all`

```
all condición lista
```

Entonces, `all` es una función que recibe como primer parámetro una `condición` (una función a ser evaluada en un solo elemento que retorna un booleano) y como segundo parámetro una `lista` de esos elementos.

Volviendo al ejercicio, la condición era que una nota (o sea, un número) sea mayor o igual a cuatro, y la lista era una lista de notas (o sea, una lista de números).

```
estaAprobado unAlumno = all esNotaAprobada (snd unAlumno)
```

La condición la representamos por una función que recibe un elemento de la lista y devuelve un Bool

Tenemos una función (`all`) que recibe otra función como parámetro (`esNotaAprobada`).

A las funciones que reciben funciones como parámetro (o devuelven funciones) las vamos a llamar funciones de orden superior.

Nota: en funcional el not no es una función de orden superior porque recibe como parámetro un Bool y devuelve un Bool

Ejemplo ReReloaded

Necesito una función que dado un alumno me diga su nombre y otra que me diga sus notas

```
nombreAlumno = fst
notasAlumno = snd
```

También estaría bueno tener una función que dado un alumno me diga la cantidad de notas que tiene

```
cantidadNotas unAlumno = length (notasAlumno unAlumno)
```

Tenemos una función constante definida de la siguiente manera

```
cursoK9
= [("mica",[8,6,9,10,8]),("pepa",[2,1,3]),("caro",[6,7,9]),("beto",[10,9,2,8])]
```

Entonces, así como está `all` hay otras funciones de orden superior copadas que reciben una función (a aplicar en cada elemento de una lista) y la lista.

- Qué pasa si queremos conocer una lista con los alumnos de un curso que dieron parcialitos (o sea, que tienen más de 3 notas)?

```
quienesDieronParcialito unCurso = ...
```

`quienesDieronParcialito unCurso = filter` ***unaCondición*** `curso`

unaCondición debería ser una función que recibe un elemento de la lista curso (o sea, un alumno) y me devuelve un booleano.

`quienesDieronParcialito curso = filter` **`((3<).cantidadNotas)`** `curso`

((3<).cantidadNotas) :: Int → Bool

((3<).cantidadNotas) es una sola función (ok, está compuesta por la función (3<) y cantidadNotas pero yo uso la composición de dichas funciones que es UNA SOLA FUNCIÓN)

Qué hace **filter**? Filter recibe una condición y una lista L. Devuelve una nueva lista con los elementos de L que cumplan la condición

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

Si a filter le mando una lista L de **n** elementos me va a devolver una nueva lista que tiene una cantidad de elementos **menor o igual a n**
Si a filter le mando una **lista de elementos de un tipo a** me va a devolver una nueva **lista de elementos del tipo a**

- Qué pasa si queremos conocer los nombres de todos los alumnos de un curso?

```
nombres unCurso = map nombreAlumno unCurso
```

Qué hace **map**? Map recibe una "transformación" y una lista L. Devuelve una nueva lista con los elementos resultantes de aplicar la transformación a cada elemento de L

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

Si a map le mando una lista L de **n** elementos me va a devolver una nueva lista que tiene una cantidad de elementos **igual a n**
Si a map le mando una **lista de elementos de un tipo a** me va a devolver una nueva **lista de elementos del tipo b (siendo el tipo b lo que devuelve la transformación)**

- Qué pasa si queremos conocer los nombres de los alumnos de un curso que estén aprobados?

```
nombresAlumnosAprobados unCurso = map nombreAlumno (filter estaAprobado unCurso)
```

Primero obtenemos una lista con los alumnos aprobados (una lista de tuplas) y despues obtenemos una lista de strings (lista de nombres) aplicando la transformación nombreAlumno a cada elemento de la lista de alumnos aprobados.