Aplicación parcial y currificación

¿Y cómo es que ocurre esa magia de la aplicación parcial? ¿Por qué funciona eso de pasar menos argumentos que los que le corresponden a la función?
¡Gracias a la currificación!
¿Gracias a lo qué?

La currificación es una forma de definir las funciones de forma de considerar a una función como una secuencia de funciones de un único argumento.

¿Cómo es eso?
Las funciones reciben un único parámetro, el primero, y devuelven otra función equivalente pero con un parámetro menos.

Ej.: Según sabemos, la función `map` tiene 2 argumentos: una función de conversión y una lista, y devuelve una nueva lista correspondiente al resultado de aplicar la función a cada elemento de la lista.

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

Pero en realidad, lo que pasa es que la función `map` recibe una función de tipo `(a -> b)` y devuelve otra nueva función, la cual a su vez recibe una lista de tipo `[a]` y devuelve otra lista de tipo `[b]`. O sea, el tipo anterior es equivalente al siguiente:

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

Viendo lo mismo en una aplicación concreta, se puede hacer:

```
map length ["Hola","mundo"]
```

Y es lo mismo que:

```
(map length) ["Hola","mundo"]
```

Ya que `map length` es el resultado de `map` "con su único parámetro instanciado", y es una función de tipo `[a] -> [b]`.

Entonces, gracias a la currificación podemos hacer cosas como `(2 *)`, siendo que en realidad la función `(*)` es una función de 2 argumentos.

En C y en Pascal yo no tengo esa posibilidad: `mayor(8, 4)` no se puede descomponer.
En Haskell yo también podría definir una función de esa manera, recibiendo una tupla en lugar de dos argumentos:

```
mayor :: (Int, Int) -> Int
mayor (a, b) | a > b     = a
             | otherwise = b
```

De esta manera la function queda sin currificar. La principal desventaja: ya no puedo aplicarla parcialmente.
Pueden chusmear en el Prelude dos funciones:

- `curry`
- `uncurry`

Empezando a aplicar el término:

- ¿Qué es *currificar* una función? Hacer que la misma tenga `n` argumentos en lugar de un argumento solo, que sería una tupla de n elementos.

Ejemplo:

Función sin currificar `map :: ((a -> b), [a]) -> [b]`

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

Que se asocia a derecha:

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

- currificar una función permite poder aplicarla parcialmente.

Inferencia de tipos y polimorfismo

En algún momento dijimos que nuestras funciones tienen un tipo, y que al definirlas en Haskell podíamos explicitar este tipo o no. En el caso de que no, dijimos que el motor del mismo Haskell se encarga de resolvernos este problema.

Pero ahora, como somos curiosos, queremos entender un poco cómo demonios funciona en Haskell esto que llamaremos inferencia de tipos.

funcion id: `id x = x`

¿Qué conozco de `x`? Que tiene que ser "algo".
¿Qué significa? que el parámetro puede ser de cualquier tipo.
¿Entonces cuál es el tipo de `id`?

```
id :: a -> a
```

En particular diremos que esta función es *polimórfica* a sus parámetros, porque podemos pasarle parámetros de distinto tipo. En este caso el **polimorfismo es paramétrico** porque no tengo restricción con respecto al tipo del parámetro que puedo pasarle.

Otras funciones polimórficas (paramétricas): la función length

```
length [] = 0
length (x:xs) = 1 + length xs
```

¿Qué conozco sobre el parámetro? Que debe ser una lista.
¿Algún tipo de lista en particular? No.
Entonces ¿cuál es el tipo de `length`?

```
length :: [a] -> Int
```

Otro caso:
las funciones `fst` y `snd`

```
fst (a,_) = a
snd (_,b) = b
```

¿Qué tipos tienen?

```
fst::(a,b) -> a
```

```
snd::(a,b) -> b
```

Otras funciones que vimos que tienen polimorfismo paramétrico: `head` y `tail`

Ahora, tenemos la definición de la función `sum`:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

¿Qué conozco del parámetro? Que tiene que ser una lista, por el pattern matching...
¿Y qué mas? Y… que sus elementos ¡¡tienen que poder sumarse!! Estamos pasando `x` como parámetro de la función `(+)`, entonces decimos que tiene que poder ser parámetro de esa función.
Entonces *¿cuál es el tipo de `sum`?*

```
sum :: [Int] -> Int
```

¡No, no!
¿qué pasa si hago?

```
sum [1.2, 3.4] ?
```

¡Funciona! (4.6) ☺

¿Y si hago `sum ["hola","pianola"]` ?

Ahí sí, se rompe.

Entonces acá aparece lo que llamaremos **restricciones de tipo**: nuestra función no recibirá un valor de un tipo en particular ni tampoco un tipo genérico. A esto lo llamamos **polimorfismo ad-hoc** porque puede recibir un valor dentro de un conjunto determinado de tipos.

bueno si hacemos `:t sum` nos dice

```
sum :: Num a => [a] -> a
```

`Num` nos restringe el tipo de 'a' a todo aquel tipo que pertenezca a la clase `Num`. Entonces sí, entendieron bien, `Num` es una Clase. Muchas veces lo van a ver nombrado como type class, porque justamente es una clase de tipo.

¿Y eso que quiere decir? Que `Num` me define un **contrato (o interfaz),** que es el conjunto de operaciones que le puedo pedir. Por ejemplo, a un `Num` le puedo pedir:

+ , - , *, abs, negate, fromInt, signum (el signo)

```
sum [] = 0
sum (x:xs) = x + sum xs
```

¿Pero eso no se lo puedo pedir también a un `Int`?
Bueno, resulta que `Int` por ser un tipo en particular que soporta las operaciones definidas en `Num` , lo vamos a llamar *instancia* de la clase `Num`.

Un `Float` también es una instancia posible de `Num`, eso me permite que sum no esté restringida a sumar listas de enteros, sino que también funcione para listas de números decimales (floats).

Para pensar entre todos:

¿Cuál es la ventaja de definir …
`sum :: Num a => [a] -> a`
... en lugar de...
`sum :: [Int] -> Int` ?

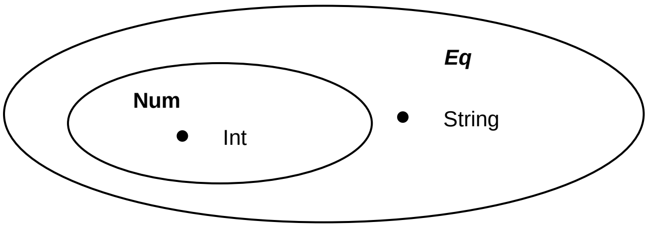
Que mi función acepta parámetros de distinto tipo sin necesidad de definir varias veces la misma función. Me abstraigo del tipo real usando una clase que termina siendo "una interfaz" (vamos a ver este mismo concepto en Objetos).

Otro ejemplo: función `elem`

```
elem _ []     = False
elem x (y:ys) = x == y || elem x ys
```

¿Qué pasa con los parámetros? mmm se comparan. Entonces si se comparan `(==)` el tipo del parámetro tiene que tener definida la comparación. Llamaremos a estos tipos `Eq`.
Otro ejemplo:
`parIgual (i,d) = i == d`

Un `Int`, ¿es un `Eq`? Sí, porque puedo "igualarlos". Pero también es un `Num`. Gráficamente:



Para pensar:

Un `Num` (Número), también es un `Eq`. Entonces `Int` es tanto un `Eq` como un Número.
La diferencia es que un `String` también es un `Eq`, pero no es un Número.

Entonces, ¿qué podemos decir sobre inferencia de tipos? Que tenemos que prestar atención a los parámetros que recibe la función y ver qué funciones u operaciones se aplican sobre ellos.

La clase `Ord`

Esta clase define operaciones para determinar si un elemento es menor o mayor que otro (sirve para **ord**enar elementos).

Un número… se puede ordenar: la clase `Num` es un `Ord` y un `Eq`. Entonces un `Int` por pertenecer a un `Num` puede:

- sumarse con otro (por pertenecer a `Num`)
- responder si es mayor o menor que otro (por pertenecer a `Ord`)
- responder si es igual o distinto a otro (por pertenecer a `Eq`)

Un `String`… se puede ordenar: "alba" < "alma". Entonces un `String` es tanto un `Ord` como un `Eq` (por lo que vimos recién).

Si

```
min a b | a > b     = b
        | otherwise = a
```

¿De qué tipo es min?

```
>:t min
min :: Ord a => a -> a -> a
```

Combinamos con aplicación parcial, *¿de qué tipo es min 5?*
Podríamos pensar que `min 5 :: Ord a => a -> a`
Pero en realidad al evaluar (`min 5`) le estoy pasando un número… entonces el segundo argumento tiene que ser del mismo tipo (por la definición de `min`, va de `a` en `a` en `a`)… entonces no debería sorprendernos que:

```
> :t min 5
min 5 :: (Ord a, Num a) => a -> a
```

Para ver en casa: *¿de qué tipo es min "hola"?*

Indicar cuáles son los tipos de esta función

```
detect criterio = head . filter criterio
```

Esto tiene una trampita. Filter criterio nos devuelve una función que espera una lista, pero esa lista está implícita en la definición.

```
detect criterio xs = (head . filter criterio) xs = head (filter criterio xs)
```

¿Cómo nos damos cuenta? Porque `filter` necesita una función y una lista.
Entonces sabemos que:

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

g f a b  = f a == f b
g :: Eq b => (a -> b) -> a -> a -> Bool

h a b    = length (fst a) - length (snd b) + (snd a) - (fst b)
h :: Num b => ([a], b) -> (b, [c]) -> b
```

en realidad como `length` devuelve un `Int`, `b` no es un `Num` sino un `Int`:

```
h :: ( [a], Int ) -> ( Int, [b] ) -> Int
```