Repaso primer módulo

¿Qué era efecto colateral? ¿Qué era transparencia referencial?
¿El efecto colateral, está asociado al lenguaje o al paradigma?
Los tipos, ¿están asociados al lenguaje o al paradigma?
Anotamos en el pizarrón:
El **paradigma** define si existe efecto colateral, si tiende a ser más declarativo/más imperativo.
El **lenguaje** puede ser tipado (es el compilador el que lo define, si soy estricto: fuertemente tipado; si no chequeo los tipos: débilmente tipado).
¿Qué indica un dominio e imagen en un programa funcional? El rango de valores posibles que acepta como input y lo que devuelve como output.

Intro a Recursividad / Pattern matching

Mecanismos de iteración: Dentro de una función, ¿puedo tener una estructura de repetición?
Sí, gracias a la recursividad: *una función que se llama a sí misma*.

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$$

Genéricamente:

$$n! \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

De hecho esta definición matemática la puedo pasar a Haskell:

(1)
```
factorial n | n == 0     = 1
            | n > 0      = n * factorial (n - 1)
```

Otra opción (2):
```
factorial 0  =  1
factorial n  =  n * factorial (n-1)
```

En (1) Cualquier valor entero “matchea” con n (si quiero evaluar factorial 3, la variable n se unifica con el valor 3). En (2) interviene el concepto de **pattern matching**: tratamos de hacer que encaje un determinado valor en una expresión:

- “factorial 3” no matchea con “factorial 0” (valor vs. valor),
- pero 3 sí es asignable a n (valor vs. variable en el sentido matemático).

Observaciones: las guardas son excluyentes. Una vez que matcheé un patrón en una guarda, no puedo entrar en otra. Este tipo de pattern matching se llama “de secuencia”, porque define un orden en el que se procesan las guardas de una función. Que haya un orden es toda una cuestión filosófica, que vamos a tratar en un par de clases.

¿De qué tipo es factorial?
Recibo un número, devuelvo otro número.
¿Qué tipo de número? Sólo enteros.
Entonces
```
Main> :t factorial
factorial :: Int -> Int
```

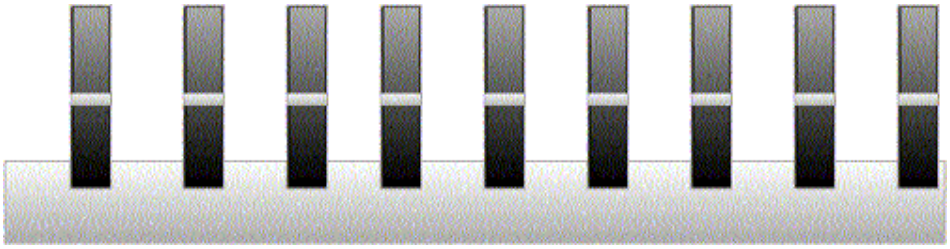
Recursividad e Inducción

El concepto de recursividad viene atado siempre al de **inducción:** defino P(0), y P(N + 1) en base a P(N). Por eso algunos autores prefieren dar el factorial de libro:

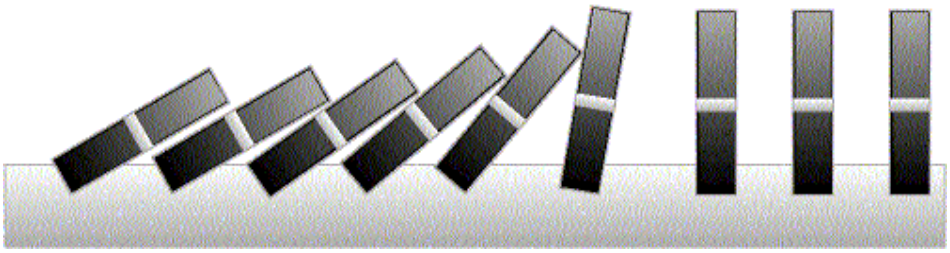
```
factorial 0         =  1
factorial (n+1)     =  (n+1) * factorial n
```

A esta estrategia de codificación se la llama (n + k) pattern matching.

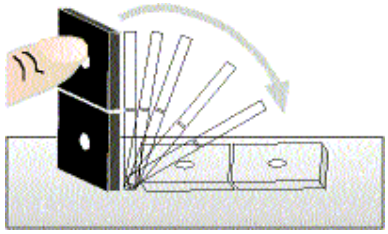
Inducción: metáfora asociada, Dominó



Nosotros sabemos que si tiramos un dominó, van a caer todos los sucesivos…



1) **Caso base:** Si presionamos un dominó en la parte de arriba, cae.



Entonces, tiramos un dominó. P(0) se cumple.

2) **Hipótesis:** Si el dominó es contiguo al que cae, caerá también.

3) **Demostración**
Si nos posicionamos en cualquier dominó P(N), P (N + 1) también caerá por estar contiguo a P(N), entonces probamos por inducción que todos los dominó van a caer[1].

Otra metáfora asociada a la inducción: la fila del cine, tenés la fila para sacar entrada y la fila para entrar a ver la película.
Llego yo, P(0) = “voy a sacar entrada al cine”
Hay un montón de gente haciendo cola, le pregunto al último: “¿es para sacar entrada para Avatar?”

[1] Metáfora extraída de
http://www-static.cc.gatech.edu/~idris/AlgorithmsProject/ProofMethods/Induction/UnderstandingInduction.html

Si P(N) es verdadero, asumo que los demás también están sacando entrada para la misma película: P(N + 1) se cumple también.

Ahora, me interesa armar una función que sume los primeros n números.
[Dejar a la clase 10' a que lo resuelva]

```
sumarPrimerosN n | n == 0 = 0
                 | n >  0 = n + sumarPrimerosN (n - 1)
```

¿Qué tipo tiene la función? Int -> Int, nuevamente.

Un ejercicio más, armar una función que me devuelva si un número es primo o no. Matemáticamente, un número es primo si es divisible sólo por sí mismo y por uno. Esto lo podemos plantear recursivamente…
[que tiren ideas]
Acá empezamos a abrir la brecha entre imperativo y declarativo…

¿Cómo sería en procedural?

```
function esPrimo(n: integer): boolean
begin
   for i ← de 2 a (n - 1)
   begin
      if (mod(n, i) = 0)
      begin
         ↑ false;
      end;
   end;
   ↑ true;
end;
```

Pero acá no tengo variables para ir asignando destructivamente (no hay efecto de lado).
¿Cómo se guardaba el valor intermedio de factorial? En n * factorial (n – 1)

Entonces, una forma de almacenar estado es… ¡pasando valores como argumentos!

```
primo n = primoAux n (n - 1)

primoAux n i | i == 1        = True
             | rem n i == 0  = False
             | otherwise     = primoAux n (i - 1)
```

¿Cómo leo el programa?
primo: Un número es primo si es primo respecto a sus números precedentes.
primoAux: Un número es primo respecto a sus números precedentes si no es divisible por alguno de sus números precedentes.

Todo muy lindo, pero… ¿quién labura?
El ***motor*** Haskell, que trabaja hasta que llega a una expresión irreductible:

```
primo 4 = primoAux 4 (4 - 1) = primoAux 4 3 = primoAux 4 (3 - 1) = primoAux 4 2 =
False.
```

Recordemos que una vez que matcheé un patrón, no puedo entrar en otro.

Observación: Esta es la única vez que les contamos qué pasa abajo… el chiste de trabajar en forma declarativa es abstraerme de cómo se termina resolviendo y concentrarme en qué es lo que hay que hacer.

Busquemos dominio e imagen de la función:
`primo`: Recibimos un `Int`, devolvemos un `Bool`.
`primoAux: Int -> Int -> Bool`