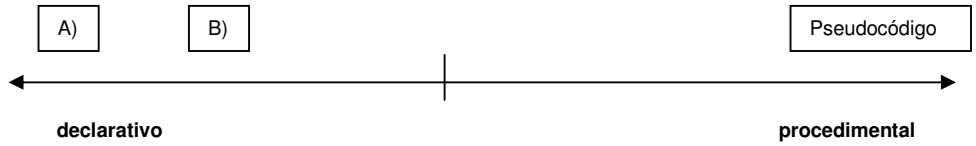
Repaso de la clase pasada

¿Qué es una función recursiva? Una función que se llama a sí misma.
¿Cómo pruebo que una solución recursiva funciona? A través de la inducción.
P(0) es importante para cortar la recursividad. Recordar el ejemplo de Santa Bernardina del Monte, de Leo Maslíah.
¿Cómo guardo estados intermedios en funcional? Pasándole argumentos a una función recursiva y relacionándolos por medio de pattern matching
Haskell, es ¿débil o fuertemente tipado? Fuerte, los tipos los infiere el intérprete. ¿A través de qué? De los operadores o funciones que se aplican sobre los argumentos, y a través de la expresión que se calcula.
¿Cómo defino una lista?
¿Cómo defino una tupla?
¿Qué diferencias hay entre lista y tupla?
Listas por comprensión: ¿qué diferencias tienen estas definiciones de la función doble, que dada una lista de números devuelve la lista con los números multiplicados por dos?

A)	B)
`dobles xs = [ 2 * x | x <- xs ]`	`dobles []     = []` `dobles (x:xs) = (2 * x) : doble xs`

Con A) ni siquiera se de qué manera accedo a la lista, que en B) se ve que es a través del matching cabeza/cola. A) es más "declarativo", el motor Haskell se encarga de resolver el cómo, yo me preocupo por definir correctamente la condición. De todas maneras, si definiéramos la misma función en pseudocódigo (como hicimos en las primeras clases), podríamos hacer la siguiente comparación:



Si yo tengo que resolver:

```
> cuadrado (3 + 4)
49
```

¿Cómo lo resuelve?

¿cuadrado 7 = 7 * 7 = 49?
¿(3 + 4) * (3 + 4) = 7 * (3 + 4) = 7 * 7 = 49?
¿(3 + 4) * (3 + 4)? = 9 + 12 + 12 + 16 = 21 + 28 = 49?

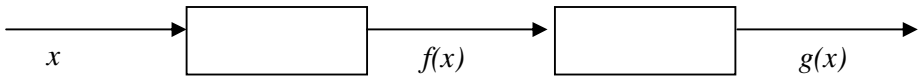
Estas diferentes formas de hacerlo corresponden a estrategias de evaluación (reducción de expresiones).
Si estoy en un paradigma procedimental/imperativo: la estrategia la decido al programar.
Si estoy en un paradigma declarativo: se lo dejo al motor.

En nuestro caso el motor es el intérprete Haskell, que reduce una expresión de una manera que nosotros no conocemos, y no nos interesa conocer (porque nuestro objetivo pasa por otro lado. Esto de concentrarse sólo en lo esencial de un problema se llama **abstracción**, y está

bueno para nosotros porque nos desentiende de los detalles de cómo se termina resolviendo). He aquí la parte buena de la declaratividad: no sólo escribo menos, sino que escribo lo que nadie puede escribir por mí. Igual vamos a volver a esto en un ratito.

Composición

¿Cuál era el concepto matemático de composición?

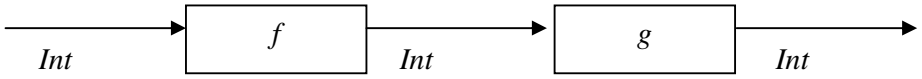


$g(f(x)) = (g \circ f)\ x$

¿Qué condición se tenía que cumplir? La imagen de f(x) tenía que coincidir con el dominio de g(x).
Bueno, exactamente igual es el concepto dentro del paradigma funcional:
La notación es (g . f) x = g (f x)

Vamos con un ejemplo sencillo:

g x = x + 3
f x = 2 x



(g . f) x = g (f x) = f x + 3 = 2x + 3

¿Qué ventaja tiene esto?
cuadruple = doble . doble

Ejemplo: decir si el enésimo elemento de una lista de números es par.
g = saber si un elemento es par
f = tomar el n-ésimo elemento de una lista.

g x = even x
f xs n = n-simo xs n donde n-simo es:

```
enesimo n (x:xs)  | n == 1    = x
                  | otherwise = enesimo (n - 1) xs
```

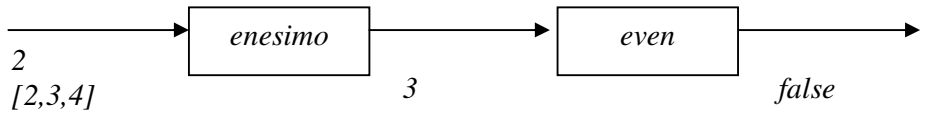
Entonces puedo definir la función

```
enesimoEsPar xs n = even (enesimo n xs) = (even . enesimo n) xs
```

Lo que sí tenemos que ver es de qué tipo es f, de qué tipo es g:
`even :: Int -> Bool` (o de un número devuelve un booleano)
`enesimo :: Int -> [Int] -> Int` (más genéricamente: la posición es un entero, la lista de números y "me devuelve" un número de esa lista).

¿Puedo componer even con enésimo[1]?

Sí, porque yo voy a recibir una lista de enteros y un entero, y lo que me devuelve f (un entero) se lo voy a pasar a g que recibe un entero. La composición de ambas funciones me devuelve un booleano:



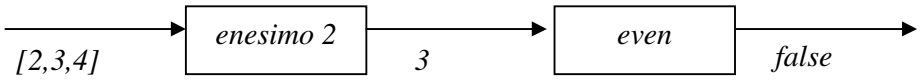
Para pensar en el colectivo:

f = (even . enesimo 1) es una función me indica si el primer elemento de una lista es par.
Si a *f* la llamo primeroEsPar:

```
Main> primeroEsPar [5, 0]
False
```

f = (even . enesimo 2) es una función me indica si el segundo elemento de una lista es par.
…y así sucesivamente.

O sea, en realidad la composición es[2]:



¡Uuuuuh! Lo repito una vez más y lo piensan para la próxima clase.

Otros ejemplos:

Tengo el siguiente requerimiento:

- quiero saber si una persona es mayor de edad

¿Cómo modelo a la persona? Con una tupla.

Ok, encontré una abstracción para representar esa **entidad**. ¿Me interesa todo sobre la persona? No, me quedo con lo que es esencial para mí y dejo una tupla con el nombre y la edad: (String, Int)

Ahora puedo separar el problema en dos partes:

1) saber la edad de una persona
2) saber si esa edad es >= 21 (ó 18)

1) y 2) las abstraigo en **funciones**.

Lo bueno es que ya se qué objetivo tiene cada función (ya se cuál es su **responsabilidad**).

[1] En realidad no puedo componer even con enésimo, pero sí puedo componer even con la función (enesimo n). Sobre esto vamos a charlar próximamente…

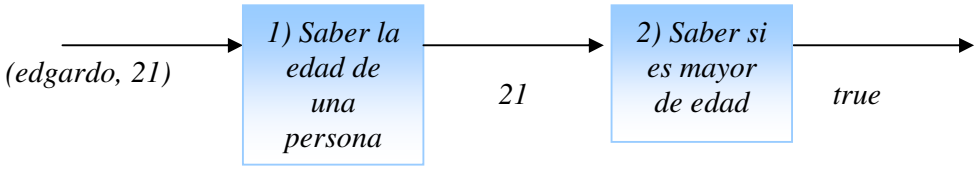
[2] **Observación:** fíjense que el orden en el cual se escriben las funciones a componer es inverso al orden en el cual se va resolviendo cada caja

Podemos pensar cuál es el input y el output de 1) y 2)

Para saber la edad de una persona)
Input: una persona → (String, Int)
Output: la edad → Int

Para saber si una edad es considerable "mayor de edad")
Input: una edad → Int
Output: true o false → Bool

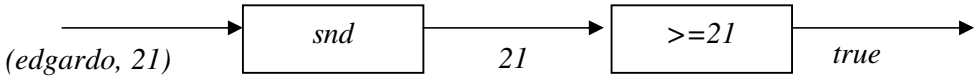
Aquí vemos que el dominio de 2) se corresponde con la imagen de 1), entonces puedo componerlas. Pensando en un ejemplo, armamos las cajitas escribiendo en castellano qué es lo que hacen[3]:



En la **interacción** de ambas funciones (mediante la composición) terminamos de resolver el ejercicio. Sólo nos queda

a) encontrar alguna función existente que pueda servirnos o
b) desarrollarlas nosotros.

Tenemos suerte, ya existen snd y (>= 21) que vienen con el Prelude…



esMayorDeEdad persona = snd persona >= 21 *Otra opción:* ((>= 21) . snd) persona

Aquí vemos que la composición se hace entre la función snd y la función (>= 21).

En definitiva, lo que acabamos de hacer es **diseñar** la solución:

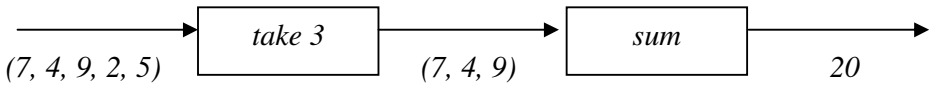
- encontramos los componentes de un problema (entidades y funciones)
  - *Corolario:* dividimos así un problema en partes
- le asignamos responsabilidades (qué debe hacer cada función)
- entendemos cómo interactúan esos componentes (en este caso, mediante composición)

Otro requerimiento:

- Quiero obtener la suma de los 3 primeros elementos de una lista de enteros. Compongo take y sum. Pero take tiene **2** argumentos, entonces en realidad estamos componiendo sum con… take 3.

[3] Está bueno pensar primero el objetivo de la función para recién después buscarle un nombre representativo y los parámetros que necesita

```
sumaDe3PrimerosElementos xs     = sum (take 3 xs)
sumaDe3PrimerosElementos ~~xs~~  = (sum . take 3) ~~xs~~[4]
sumaDe3PrimerosElementos        = sum . take 3
```



Listas infinitas revisited

¿Se acuerdan cuando hicimos
[1..]?

Esto nos crea una lista potencialmente infinita. ¿Podemos hacer algo como:
head [1..]?
¿Se puede o no?

Como estamos acostumbrados a trabajar en lenguajes con evaluación ansiosa, primero trato de reducir la expresión que tengo como parámetro y después invoco a la función. Pero Haskell (y muchos otros lenguajes del paradigma funcional) trabajan con el concepto de evaluación diferida (**lazy evaluation**), con lo que voy evaluando los argumentos recién a medida que los voy necesitando. Por eso puedo pedirle la cabeza de una lista infinita y me devuelve:
1

Para fijar el concepto: tenemos un carretel o un cable laaaaargo (5 metros por lo menos). Llega un cliente:

- *Señor, ¿me da 10 cm de cable?*
- Cómo no, ya le doy…

y empieza a desenrollar todo el cable.
El cliente dice: "*Pero señor, yo sólo quiero 10 cm…*"
Bien, desenrollar todo el cable o cortar sólo lo que me pide el cliente es la diferencia entre evaluación ansiosa y evaluación diferida.

¿Qué sentido tiene?
Tengo una función que me devuelve los números primos de una lista. ¿Cómo sería usando listas por comprensión?

```
primos xs = [ x | x <- xs, primo x ]
```

¿Cómo hago si quiero encontrar el primer primo comenzando por un número n?

1)

```
primerPrimo n | primo n    = n
              | otherwise  = primerPrimo (n + 1)
```

2) Uso la función primos

```
primerPrimo n = head (primos [n..])
```

[4] BONUS: La simplificación de términos recibe el nombre de "conversión eta" (η-conversion). Para más información puede verse el artículo http://www.haskell.org/haskellwiki/Eta_conversion y http://www.haskell.org/haskellwiki/Pointfree. La idea de esta reducción de términos es mejorar la legibilidad de los programas eliminando paréntesis y variables redundantes.

o bien `(head . primos) [n..]`

Nuevamente, la opción 2) es casi castellano: tomá el primer elemento de la lista de primos comenzando de n.

La opción 1) es más declarativa que una posible opción en pseudocódigo, pero menos que la opción 2). No es que una es mejor que la otra, sólo que una abstrae más y la otra necesita que yo le diga más cosas.
La ventaja es que con evaluación diferida pude aprovechar una función que recibía una lista y casi no necesité hacer más nada que **componer**.

Anotamos en el pizarrón: la evaluación diferida me permite trabajar con listas potencialmente infinitas. Lo enmarcamos en un cuadro y recordamos que es (o era) "una típica pregunta de final".

Si hacemos:
```
Main> take 3 [6..]
```
¿Qué devuelve?

Otro ejemplo donde se ve evaluación diferida:

```
> fst (9, 1/0)
```
Me devuelve 9. ¿Qué pasa si tengo evaluación ansiosa? División por cero. Es un ejemplo didáctico, ojo, no es que está bueno que yo ponga fruta en un argumento.

Y volviendo al ejemplo del principio:
Si yo tengo que resolver:
> cuadrado (3 + 4)
¿qué hago? ¿Resuelvo primero el argumento o invoco a la función con la expresión que me pasaron como argumento?
Efectivamente…
Si cuadrado x = x * x, no evalúo el argumento sino hasta que lo preciso: (3 + 4) * (3 + 4). Bueno, a partir de ahí ya tengo que entender cómo resuelve Haskell el (*), cosa que ya no me resulta tan interesante.

En los lenguajes imperativos estamos acostumbrados a la que las cosas funcionen en forma ansiosa, pero en funcional se nos abre una nueva puerta.

- Con la evaluación diferida sólo se evalúa aquello que realmente se necesita.
- Si una expresión puede andar, con evaluación diferida seguro que anda. Si se rompe, es porque necesité algo que se rompía.
- ¿Por qué no tengo esto en C? Por el efecto colateral. Yo solo puedo cambiar el orden esperado en la evaluación de mis expresiones si se que este cambio no va a afectar al resto del mundo.
- Porque funcional es un paradigma atemporal (no importa la secuencia en el paradigma, al menos en la mayoría de los casos). No hay un antes y un después en una función de análisis.