Tipos

- ¿Cuál es la condición para poder usar una expresión como argumento de una función?
  Que sea del tipo que la función espera.
- Al motor de Haskell, normalmente, no hace falta decirle de qué tipo son las cosas, lo infiere (¡se da cuenta solo!)
- Inferencia de tipos vs. Chequeo de tipos: el intérprete no exige que definamos los tipos de una función, pero sí chequea que los tipos del dominio y la imagen sean válidos.
- ¿Qué tipos conocemos? -> Números, Booleanos, Strings, funciones... y a partir de ahora, listas y tuplas.

Listas

Al igual que en lógico, las listas tienen una definición recursiva. Una lista está compuesta por:

- Una cabeza, y…
- Una cola, que es una lista compuesta por los elementos restantes.

P(0) = La lista que no tiene elementos no puede dividirse en cabeza y cola. Se denota [ ] y se dice **lista vacía**.
P(N) = Es una lista con una cabeza y (n–1) elementos. El operador ":" permite dividir cabeza y cola: (x:xs)

P(N+1) = Es una lista con una cabeza y n elementos.

Ejemplo: la lista con los números `[1, 2, 3]` puede escribirse:
`(1:[2,3]) = (1:(2:[3])) = (1:(2:(3:[])))`

Los paréntesis son sólo para mayor claridad sobre cómo se asocian los elementos de la lista (cabeza a izquierda y cola a la derecha, y esta última es a su vez una lista), pero en realidad no forman parte de la sintaxis de la lista. Por lo tanto, lo mismo que antes también puede escribirse de esta otra forma:
`1:2:3:[]`

Habiendo mencionado eso, cabe aclarar que muchas veces vamos a querer pasar una lista por parámetro, y para decir "tooooodo esto es UN parámetro" vamos a necesitar usar un par de paréntesis:
`> fun (1:2:3:[])`

Ejercicio: encontrar la cabeza y la cola de estas listas.

Lista	Cabeza	Cola
[1, 4]	1	[4] ojo, no es 4, es la lista compuesta por 4
[]	No existe, no se puede dividir.	
[[1, 7], [8, 7, 5], []]	[1, 7]	[[8, 7, 5], []]. Se puede usar listas de listas
[8, "hermanos"]	No es una lista posible *en un lenguaje con chequeo de tipos estático*. Una lista es un tipo de dato recursivo, compuesto por **elementos del mismo tipo**.	

¿Puede una lista ser infinita? En el lenguaje que vamos a usar, sí puede. Algunos ejemplos:

Lista	Qué es
[1..]	es la lista de todos los números naturales comenzando por el 1
[1, 3..]	es la lista de todos los números naturales impares
[1, -1..]	[1,-1,-3,-5,-7,-9, etc.]

También puedo usar la función enumFrom.

`enumFrom 2`

¿Qué sentido tiene computacionalmente? Por ahora, lo dejamos ahí.

¿Qué más podemos hacer con una lista?
`head / take / at`

`head [1, 7, 9]` devuelve la cabeza de una lista.

¿Cómo se resuelve head?
```
head (x:xs) = x
```

Como `xs` no nos interesa utilizarlo en la función, podemos hacer:
```
head (x:_) = x
```

Recuerden que todo esto está en el Prelude.hs

¿Qué tipo de lista podemos recibir? A ver, probemos:
`head ["hola" , "mundo"]` nos devuelve `"hola"`
`head [[], [1,2], [3, 4]]` nos devuelve `[]`

¿Qué hacemos, definimos n veces la misma función con distinto dominio?
Y… en la definición podemos decir que podemos recibir listas de cualquier tipo:
```
head :: [a] -> a
```

Devolverá la cabeza de esa lista.
Si es una lista de listas, la cabeza es una lista.

Take: me devuelve los n primeros elementos de una lista.
```
Prelude> take 2 ["hola" , "mundo", "loco"]
["hola","mundo"]
```

`take :: Int -> [a] -> [a]`
Recibo: un entero y una lista (de cualquier tipo), y devuelvo los n primeros elementos. ¿Cómo sería esto?

```
take n _  | n <= 0   = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs
```

At: la posición de un elemento de una lista que no tiene duplicados.

```
at xs elem = posicion xs elem 1

posicion (x:xs) elem n | x == elem   = n
                       | otherwise   = posicion xs elem (n + 1)
```

¿Cuál es el caso base? Cuando el elemento que busco es la cabeza de la lista.
¿Qué pasa si busco un elemento que no existe? Da error, y está bien que así sea:
```
Main> at [1,6,2,4] 5
Program error: …
```

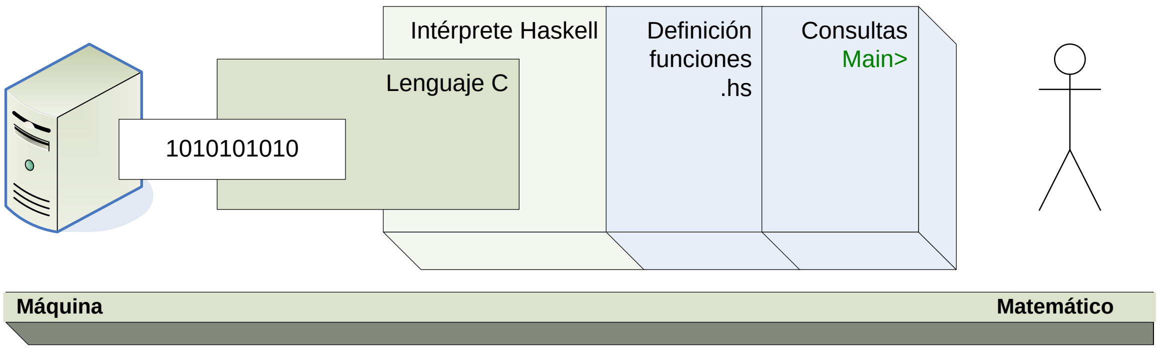
Listas por comprensión

¿Se acuerdan de la primera clase del curso?... Habíamos dicho algo así: si tengo una lista y quiero obtener los números positivos:

```
positivos xs = [ x | x <- xs, x > 0 ]
```

Responde al mismo concepto matemático de listas por comprensión.

¿Quién hace la magia? El compilador Haskell. Detrás de todo esto seguro hay saltos incondicionales, acumuladores, etc… pero este concepto me abstrae de la implementación final que corre en la máquina. Me acerco a la forma de pensar del matemático y me alejo de la máquina.



Otro ejemplo: intersección de dos listas. La intersección de dos listas la defino como: los elementos que pertenecen al primer conjunto y al segundo y son iguales.

```
interseccion xs ys = [ x | x <- xs, y <- ys, x == y ]
```

Otra forma de resolverlo:

```
interseccion xs ys = [ x | x <- xs, elem x ys ]
```

elem es una función que dice si un elemento forma parte de una lista.

Ejercicio: devolver el factorial de los números positivos de una lista.

```
factorialPositivos xs = [ factorial x | x <- xs, x > 0 ]
```

Ahora, yo no se el factorial de qué número estoy mostrando, me gustaría que apareciera un par que me dijera: numero x, factorial de x.
¿Cómo lo hago?

```
factorialPositivos xs = [ (x, factorial x) | x <- xs, x > 0 ]

Main> factorialPositivos [1, 3, -2, -1, 1]
[(1,1),(3,6),(1,1)]
```

Este concepto (par) se llama **tupla.**

Una tupla (x_1, x_2.. x_n) tiene tipos t_1, t_2… t_n

Algunos ejemplos de tuplas: ¿De qué tipo son?

Tupla	Tipo de la tupla
`(1, [2], 3)`	`(Int, [Int], Int)`
`('a', False)`	`(Char, Bool)`
`((1,2),(3,4))`	`((Int, Int), (Int, Int))`

Permite representar un tipo de dato compuesto, pero con elementos que pueden ser de distinto tipo. El número de elementos es fijo (siempre el mismo).

Comparamos tuplas con listas:

- Las listas requieren que todos los elementos sean homogéneos (no podemos mezclar en una misma misma lista números y strings).
- El número de elementos de una lista es variable, puede ser infinito.
- La lista es un tipo de dato recursivo, la tupla no, aunque ambos son compuestos.

Algunas funciones para hacer en el pizarrón en conjunto:

```
sumaPar (a, c) (b, d) = (a+b, c+d)
minPar (a,b) = min a b
```

Hay funciones estándares para separar una tupla de dos elementos:

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

¿De qué tipo son?

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

Queda para la práctica:

Ejercicios con listas:

- longitud
- suma
- concatenación
- reversa: tratar de resolverlo usando listas por comprensión y recursivamente (está bueno para mostrar que no siempre una alternativa es mejor que otra, depende del problema a resolver).