IDAÑEZ, LUCIA MARÍA

Paradigmas de Programación

Abstracción:

Un programa es una entidad muy compleja y es difícil abarcarlo totalmente, para ello necesitamos herramientas que nos permitan manejar esa complejidad, éstas herramientas se denominan **abstracciones**.
Las **abstracciones** nos permiten interpretar y conceptualizar lo que nos resulta más importante de un programa, sin tener en cuenta todos sus detalles.
(más info.: https://wiki.uqbar.org/wiki/articles/abstraccion.html)

Declaratividad:

Contraposición a la imperatividad. Importa más el "¿Qué quiero que suceda?" en vez del "¿Cómo solucionaremos esto?".
La **declaratividad** es una forma de abstracción. Consiste en no describir cómo hacer algo sino enunciar qué quiero que pase y que alguna construcción más elaborada lo resuelva por mí.
En un programa construido de forma declarativa se produce una separación entre la descripción del problema y los algoritmos o estrategias para encontrar la solución (estos son provistos por el motor).
Un programa declarativo separa claramente los siguientes elementos:

- El objetivo;
- El conocimiento;
- El motor que manipula el conocimiento para lograr el objetivo deseado.

Expresividad:

La **expresividad** de un código se define sobre las cuestiones que hacen que éste sea más fácil de ENTENDER por una persona.
Para lograr que nuestro código sea expresivo debemos:

- Usar buenos nombres
- Usar buenas abstracciones en general (y en particular, la declaratividad).
- Identar correctamente el código: separar con espacios, hacer que el código sea legible, etc.

Paradigma Funcional

Haskell (GHCI)

PILARES → APLICACIÓN PARCIAL, ORDEN SUPERIOR Y COMPOSICIÓN

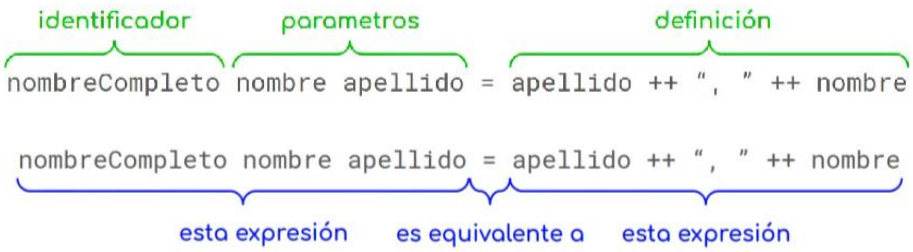
El **PARADIGMA FUNCIONAL** se basa en las funciones.
Función --> Relación (conjunto de pares ordenados) entre elementos de dos conjuntos que cumple con existencia y unicidad.
La función se asocia al concepto de CAJA NEGRA, es decir no nos interesa lo que sucede dentro, solo nos concentramos en lo que entra y sale.
En el paradigma **funcional** pensamos los programas como series de transformaciones/ transición de estados/ puentes entre tipos de estados. Pensamos a las funciones como transiciones de parámetros.

> **Transparencia referencial**: el producto de ejecutar una función es solamente su resultado y depende únicamente de los parámetros, ante los mismos parámetros produzco siempre el mismo resultado.

Un programa en el paradigma funcional es un conjunto de transformaciones.

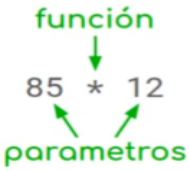
IDAÑEZ, LUCIA MARÍA 😸

Primer ejemplo de código Haskell:



Sintaxis para llamar una función:
Funciones que se definen con un nombre alfanumérico (caracteres) ->PREFIJA

->INFIJA



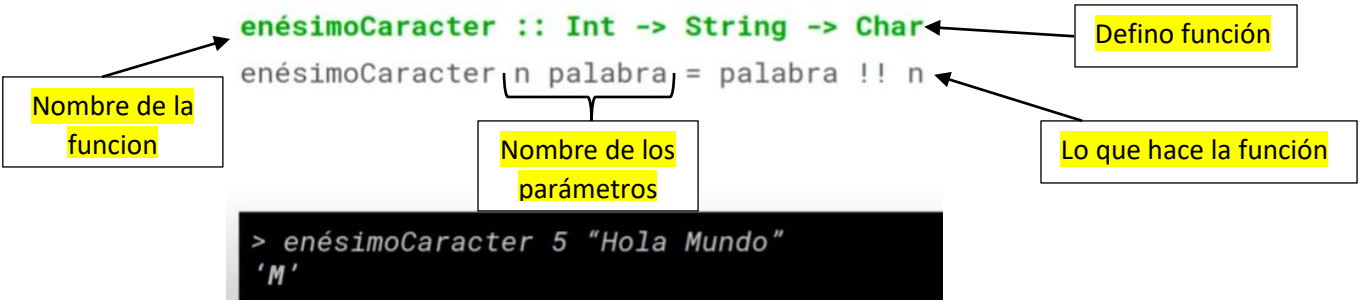
Haskell nos deja ejecutar de manera prefija o infija cualquier tipo de función.

Pattern Matching:

Nosotros podemos definir una función donde en vez de ser una variable como parámetro ponemos patrones, es decir **formas específicas que puede tomar un parámetro, que solo se van a ejecutar cuando nosotros pasemos ese parámetro**.
El PATTERN MATCHING nos indica que aquello que se nos pase por parámetro debe tener UNA FORMA/ESTRUCTURA específica.
El orden de los patrones es importante ya que Haskell lee de arriba hacia abajo.

Tipado:

Análisis de los tipos, es decir el análisis de los conjuntos de valores en el cual podemos aplicar un conjunto de operaciones.
El tipo de una función es el tipo de sus parámetros y el tipo de sus retornos.
Funcional depende mucho de hacer transiciones ya que la transición es entre un dominio y una imagen.
La sintaxis de una función es poner "nombre de la función" :: parámetro -> parámetro -> …
Hay tantas flechas como parámetros haya. El último parámetro le corresponde al resultado de la función.




```
> enésimoCaracter 5 "Hola Mundo"
'M'
```

IDAÑEZ, LUCIA MARÍA

En Haskell se pueden componer funciones de dos maneras:

Supongamos que tenemos una función “doble” que se define de la siguiente manera doble **unNumero** = 2 * **unNumero** y deseamos una función “cuádruple”, se realiza de la siguiente manera:

1) cuadruple = doble (doble unNumero) → se compone la función como en AM
2) cuadruple = (doble . doble) unNumero → ese “ . ” es como el “o” de fog(x)

Inferencia:

Haskell tiene un motor que deduce el tipo de dato en las funciones, es decir es capaz de inferir el tipo de una función.

Tipeamos “ :t *nombre de función* ”

```
f :: Bool -> Bool -> Bool
f x y = x && not y
```

Tipeo de nuestra función

:t → da el tipo
:r → recarga el archivo
:i → da la información de la restricción

```
> :t f
f :: Bool -> Bool -> Bool

> :t f True False
f True False :: Bool
```

Deducción de Haskell

Variables de Tipo:

Van a funcionar de acuerdo con el parámetro con que lo ejecutemos. Nos van a permitir definir funciones que admitan diferentes tipos.

Se definen:

```
id :: a -> a
id x = x
ignorarElPrimero :: a -> b -> b
ignorarElPrimero x y = y
```

Sin embargo, tienen algunas restricciones como las operaciones algebraicas ya que, por ejemplo, no podemos sumar/ restar/ multiplicar dos True o dos caracteres.

Type Classes:

Una Type Class es una restricción para una variable de tipo.

Estas se definen de la siguiente manera:

Le ponemos una restricción a ‘a’, es decir el tipo de “doble” es de ‘a’ en ‘a’ siendo ‘a’ un NUMÉRICO

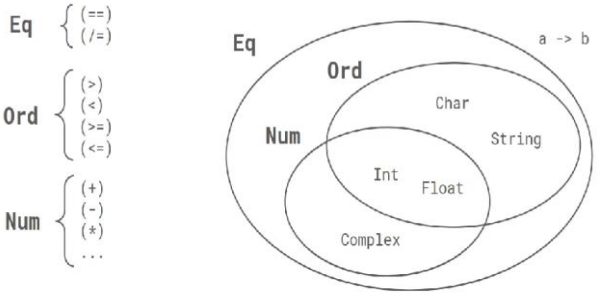
“doble” va de ‘a’ en ‘a’ siendo ‘a’ una variable de tipo

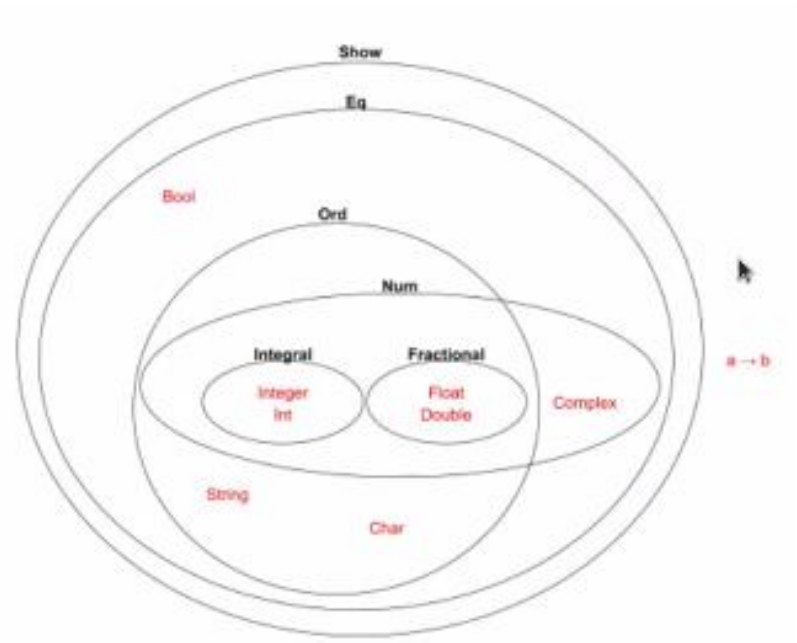
```
doble :: Num a => a -> a
doble x = x * 2
```

Para entrar en los tipos tienen que cumplir con ciertas funciones.

IDAÑEZ, LUCIA MARÍA 😸

Los tipos de clases son:
Eq -> Equiparables, aquellos que se pueden trabajar con las funciones == (igual) y /= (distinto). Todos los **Eq** son **Ord** pero no todos los **Ord** son **Eq**.
Ord -> Ordenables, los cuales además de saber si son iguales o distintos, también pueden trabajarse con >, <, >= y <=.
Num -> Numéricos, aquellos que se pueden operar algebraicamente.
Show-> se usa para todas las cosas que tienen una cadena de caracteres, las cosas que se saben mostrar por consola. `show :: Show a => a -> String`





Ejemplos de funcional:

Hallar el MÁXIMO entre tres números

```
elMayorDeLosTres :: Ord a => a -> a -> a -> a
elMayorDeLosTres x y z = x `max` y `max` z
max:: Ord a => a -> a -> a
```

Primero defino el tipo de la función, como deseamos hallar un máximo necesitaremos el tipo Ord, luego definimos lo que hará nuestra función, es decir comparar los tres ordenables mediante una función **max** que ya está determinada en el motor de Haskell.

Primero pensamos en qué necesitamos y luego constituir la respuesta en funciones más chiquitas.

IDAÑEZ, LUCIA MARÍA

Orden de precedencia de los operadores:

Precedencia (Mayor numero, mayor precedencia)	"Operador"
10	Aplicacion prefija
9	.
8	^
7	*,/
6	+,-
5	:
4	==, /=, <, <=, >, >=
3	&&
2	||
1	$

```
unaFuncion parametro1 parametro2 ... parametroN
```

Clase 13/4

Composición -> (f . g) x = f (g x)

En paradigmas, la composición es una operación que me deja construir funciones a partir de otras funciones.

Tipo de la composición: (.) :: (b -> c) -> (a -> b) -> (a -> c)

Aplicación parcial

Las funciones son valores que se pueden retornar, asociar a una variable, pasar por parámetro, etc.

Cuando aplico una función con menos argumentos de los que debería nos devuelve otra función "<function>". Esto se llama **aplicación parcial**.

La **aplicación parcial** es aplicar una función con menos argumentos de los "normales" para obtener otra que reciba los faltantes.

La aplicación parcial se lleva muy bien con la composición ya que la composición espera recibir un solo valor y nosotros podemos lograr eso con la aplicación parcial.

Un ejemplo de esto:

En vez de

```
dobleDelSiguiente = doble.siguiente
siguiente x = x + 1
doble x = x * 2
```

Podríamos escribir

```
dobleDelSiguiente = doble.siguiente
siguiente = (+1)
doble x = x * 2
```

Aún más corto

```
dobleDelSiguiente = doble.(+1)
doble x = x * 2
```

Parametrizar: recibir por parámetro

$ -> toma una función y un valor, aplica esa función con ese valor

flip -> toma una función con sus parámetros y aplica la misma función, pero con sus parámetros dados vuelta

IDAÑEZ, LUCIA MARÍA

En Haskell, todo lo que escribimos con EXPRESIONES, es decir que tienen valor, pero no tienen efecto, no hay algo que “cambie” nuestro mundo

Orden superior

Las funciones de orden superior son aquellas que toman a otras funciones por parámetros.
Ejemplos: (.), ($) y flip.

Clase 20/4

Valores Compuestos

En funcional **no podemos cambiar el mundo**, ya que tenemos valores, no variables. Es decir, podemos mostrar la imagen (el resultado de aplicar las funciones a los valores), pero no podemos cambiar el dominio.

Tupla

Una “tupla” sirve para agrupar datos simples, entre paréntesis y separándolos con comas. El tipo de una tupla es igual al tipo de sus elementos agrupados.
¿Cómo sacamos los valores que tiene dentro la tupla? Teniendo en cuenta que sea una tupla de 2 valores, utilizamos las funciones **fst** y **snd**.

Tipos de fst y snd:
fst :: (a,b) -> a
snd :: (a,b) -> b

```
saludar :: (String, Int) -> String
saludar persona = "Hola " ++ fst persona
```

En caso de que tengamos más elementos en una tupla, deberemos construir funciones que tomen los valores.
En este caso tenemos una tupla de tipo (**String**, **Int**) la cual representa el nombre y edad de una persona. Entonces, para construir una tupla haremos

Esta función va de **tupla** a **tupla**, cada tupla representa una persona

Tomamos a la tupla como una **persona**

```
cumpleaños :: (String, Int) -> (String, Int)
cumpleaños persona = (fst persona, snd persona +1)
```

Resultado:

```
*Lib> cumplirAños juli
("Julian Berbel Alt",28)
```

En este caso, “juli” ya está definido (con 27 años) y mediante esta función simulamos (no lo cambiamos) a un “juli” que cumplió años.

Type Alias
→Mediante los TYPE ALIAS haremos que los data, tuplas u otros tipos de valores sean más entendibles sobre la información que contienen.
Para generar un alias **->** type **Persona** = (String, Int) //**Persona** es el “alias”
ACLARACIÓN: No generamos un nuevo tipo de dato, sino que ponemos un “alias”, es decir que ayuda a aumentar la expresividad.

Data

Los **data** son estructuras complejas de Haskell que nos van a permitir definir tipos compuestos a partir de otros tipos.
Estructura de una data:

```
data <Nombre del Tipo> = <Constructor> <Tipo de los Campos>
```

Definimos el **nombre** del tipo

Definimos el nombre del **constructor** del data

Ponemos los campos que queremos que tenga el data

IDAÑEZ, LUCIA MARÍA

IDAÑEZ, LUCIA MARÍA

ACLARACIÓN: El **CONSTRUCTOR** es una función con la cual vamos a poder crear los tipos de datos que estamos definiendo.
Ejemplo:

Creamos un tipo de dato **Alumno**, tiene como constructor a **Alu** (función) que va recibir los campus del alumno y retornar algo del tipo "Alumno"

Tipo de "Alu":

```
> :t Alu
Alu :: String -> String -> Int -> Alumno
```

```
data Alumno = Alu String String Int deriving (Show, Eq)
```

Para construir un alumno es necesario agregar esta parte la cual establece que los tipos Show y Eq permiten que el dato Alumno esté incluido en este conjunto.

Ese agregado del final sirve para hacer que los data se puedan mostrar por consola y comparar por igualdad.
¿Cómo sacamos los valores que tiene dentro el data? Para acceder a los campos de los data lo hacemos mediante Pattern Matching:
EJEMPLO: Queremos acceder a la nota del alumno.

```
nota :: Alumno -> Int
nota (Alu elNombre elLegajo laNota) = laNota
```

Definimos el tipo de la función, esta misma es de **Alumno** a **Int**

*

Declaramos el **constructor** (Alu) y luego TODOS los campos que tiene nuestro data

Colocamos el campo que queremos extraer del data

El patrón responde a la sintaxis con que construimos Alumno, es decir "Alu elNombre elLegajo laNota", descomponemos todo con tal de llegar al campo de nota.
Una forma de ignorar las variables es mediante el "_", llamada variable anónima.
Siempre que trabajemos con datas es necesario definir funciones de acceso para cada uno de sus campos.
Otra forma de definir los data y sus funciones de acceso.

```
data Alumno = Alu {
  nombre :: String
  legajo :: String
  nota :: Int
} deriving (Show, Eq)
```

Entre llaves definimos los campos que queremos que tenga el data y a su vez se definen las funciones con las que vamos a acceder a cada campo.
Accedemos a **Int** mediante la función **nota**.

Estos son los nombres de las funciones con los que vamos a acceder a cada campo del data

*Cuando hagamos Pattern Matching con un data en una función, declaramos el CONSTRUCTOR de ese data y luego sus campos, ya que el CONSTRUCTOR nos indica la ESTRUCTURA de data

IDAÑEZ, LUCIA MARÍA

Para evitar errores en campos los cuales sus valores son limitados podemos crear un tipo de dato aparte de ese valor, de esta manera evitaremos errores.
EJEMPLO:

```
data Persona = Persona {
    nombrePersona :: String,
    edadPersona   :: Int,
    equipoArgentinoDePrimeraA :: String
} deriving (Show)
```

Tenemos este data, para evitar errores ortográficos en el campo de "equipoArgentinoDePrimeraA" vamos a crear un tipo de dato aparte.

```
data EquipoArgentinoDePrimeraA = Boca | River | SanLorenzo | Independiente | Racing deriving (Show)
```

Creamos el data con su nombre, la diferencia va a estar en que este data va a ser como una lista de **constructores**.

Así queda el data:

```
data Persona = Persona {
    nombrePersona :: String,
    edadPersona   :: Int,
    equipoArgentinoDePrimeraA :: EquipoArgentinoDePrimeraA
} deriving (Show)
```

Así queda un data declarado:

```
juli :: Persona
juli = Persona "Julian Berbel Alt" 27 SanLorenzo
```

Listas

Para cualquier tipo podemos crear una lista poniendo los elementos entre corchetes y separados por comas. Las listas solo pueden tener un tipo, es decir solo enteros, solo strings, etc.
En listas se puede:
->comparar;
->concatenar (**++**);
->voltear (**reverse**);
->saber su longitud (**length**);
->saber si está vacía ([ ] -> lista vacía);
->que nos dé una cantidad de elementos (**take** ¿?);
->descartar elementos (**drop** ¿?);
->filtrar por una condición (**filter** condición lista)

odd -> impares
even -> pares

```
filter :: (a -> Bool) -> [a] -> [a]
"Filtrado selectivo"
```

->mapear, es decir **map** aplica una función con el mismo dominio que el tipo de la lista y nos devuelve otra lista con el tipo de la imagen de la función.

```
map    :: (a -> b) -> [a] -> [b]
"Transformación elemento a elemento"
```

EJEMPLO:

```
*Lib> map nombrePersona (filter ((>30) . edadPersona) [juli, fede])
["Federico Alfredo Scarpa"]
```

Aplicando **map** tenemos, como función, **nombrePersona** y la lista de Personas con edad mayor a 30

Tenemos una lista de **Personas**, le aplicamos **filter** con la función compuesta **((>30).edadPersona)** que tiene la condición (>30) y nos da como resultado la lista de Personas con edad mayor a 30

Clase 27/4

Currificación

Visión del paradigma funcional → En Haskell todas las funciones pasan por un proceso de currificación, es decir el proceso de transformar una función que toma N parámetros en una cadena de N funciones que toman un solo parámetro.

```
f :: a ->(b ->(c -> d))
```

Expresiones Lambda

Formas de crear funciones:

→Definición normal de una función;

→Composición;

→Aplicación parcial;

→Expresiones Lambda.

Las expresiones Lambda con funciones anónimas que se usan una sola vez y luego se descartan. Se declaran de la siguiente manera:

```
( \ <parametros> -> <expresión> )
```

Parámetros de la función

Lo que realiza la función

Condiciones para el uso de expresiones lambda:

→No vamos a volver a usar la función;

→No tenemos un buen nombre para la abstracción.

Funciones definidas por partes / Guardas

Las **guardas** nos permiten definir una función por partes.

Pensemos en un ejemplo:

```
data Alumno = Alumno {
  legajo :: String,
  plan :: Int,
  notaFuncional :: Nota,
  notaLogico :: Nota,
  notaObjetos :: Nota
} deriving (Eq, Show)

data Nota = Nota {
  valor :: Int,
  detalle :: String,
} deriving (Eq, Show)
```

Tenemos el data de un Alumno de Paradigmas y queremos saber si ese alumno aprobó la materia de acuerdo con el año en que se anotó (plan).

Pensamos la función:
aprobado-> nos va a decir si el alumno está aprobado teniendo en cuenta el año en que se anotó
(nota final es la nota de los tres parciales sumados divido 3)

```
aprobado :: Alumno -> Bool
aprobado alumno {con un plan >= 1995} = notaFinal alumno >= 6
aprobado alumno {con un plan < 1995}  = notaFinal alumno >= 4
```

Esto no es válido, pero más o menos es lo que intentamos decir, es decir aquellos alumnos con un **plan** mayor a 1995 aprueban con una nota mayor 6 y aquellos alumnos con un **plan** menor a 1995 aprueban con una nota mayor a 4.
Entonces, ¿cómo resolvemos esto? Con **guardas** (|) que forman parte del Pattern Matching.
Su sintaxis es la siguiente:

```
aprobado :: Alumno -> Bool
aprobado alumno | plan alumno >= 1995 = notaFinal alumno >= 6
aprobado alumno | plan alumno < 1995  = notaFinal alumno >= 4
```

Estas son las condiciones que queríamos poner, esto da True si el alumno cumple con ellas. Entonces lo que está después del = es aquello que nos resuelve la función.

Si esta expresión booleana da TRUE entonces se ejecuta lo siguiente.
Si esa expresión da FALSE, ignora la sentencia y continua con la siguiente

Cada sentencia aplica una guarda, ese | tiene a su derecha una condición que al ser evaluada da como resultado un booleano.
Con "otherwise" como sentencia, siempre nos da True y nos sirve para cualquier otro caso que busquemos cubrir.
Definición local --> se usa dentro de una función mediante "where" y a continuación la función. Dado que es una función definida de manera local, no es necesario pasarle los parámetros ya que los puede usar directamente.
Ejemplo:

```
tipoRaices :: (Ord a, Num a) => a -> a -> a -> String
tipoRaices a b c
  | discriminante == 0 = "Raiz doble"
  | discriminante  > 0 = "Dos raices simples"
  | otherwise          = "Dos raices imaginarias"
  where
    discriminante = b^2 - 4 * a * c
```

IDAÑEZ, LUCIA MARÍA

CLASE 4/5

FUNCIÓN DE ORDEN SUPERIOR

```
data Alumno = Alumno {
  legajo :: String,
  plan :: Int,
  notaFuncional :: Nota,
  notaLogico :: Nota,
  notaObjetos :: Nota
} deriving (Eq, Show)

data Nota = Nota {
  valor :: Int,
  detalle :: String,
} deriving (Eq, Show)
```

EJEMPLO: Queremos subir la nota numérica de un alumno
->Para ello construimos una función que recibe una Nota y retorna una Nota.

```
subir :: Nota -> Nota
subir nota = nota{ valor = valor nota + 1 }
```

Usamos la variable **nota**, es el mismo dato que recibimos como parámetro

Entre llaves nos permite pasar solamente los cambios

Esta sintaxis sirve generar una copia con algunas modificaciones.

RECURSIVIDAD

Una función recursiva es aquella que en su definición se invoca a sí misma. La misma, por lo general, cuenta con una definición recursiva y al menos un caso base que corta la recursividad.

RECURSIVIDAD CON LISTAS

Las listas en Haskell son estructuras recursivas compuestas por una cabeza y una cola.
lista - -> (x : xs) siendo **x** la cabeza y **xs** la cola que, a su vez, también es una lista y la definición de lista involucra llamar a otra lista.
El (:) es la función usada para armar listas.

```
ghci> :t (:)
(:) :: a -> [a] -> [a]
```

➔ 1 : [2, 3, 4] - - -> 1 es la cabeza y la lista es la cola

```
head' (cabeza:cola) = cabeza
tail' (cabeza:cola) = cola
```

(last) - -> devuelve el último elemento de una lista.
*([…] !! *índice*) - - > tomar por índice de la lista, es decir toma el elemento de la lista con el índice que le pasemos.*

IDAÑEZ, LUCIA MARÍA

En las definiciones **recursivas** en **caso recursivo** en cada "paso" que da se va acercando al **caso base**

Algunos ejemplos de **recursividad** son:

FUNCIONES CON DOS CASOS BASE

```
take' :: Int -> [a] -> [a]
take' 0 _      = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Establecemos los casos base

Definición recursiva

take - -> toma un entero (n) y una lista y nos retorna una lista con n elementos.

```
drop' :: Int -> [a] -> [a]
drop' 0 lista  = lista
drop' _ []     = []
drop' n (_:xs) = drop (n-1) xs
```

Establecemos los casos base

Definición recursiva

drop - -> toma un entero (n) y una lista y nos devuelve una lista sin los primeros n elementos.

FUNCIONES CON UN CASO BASE

```
length' :: [a] -> Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

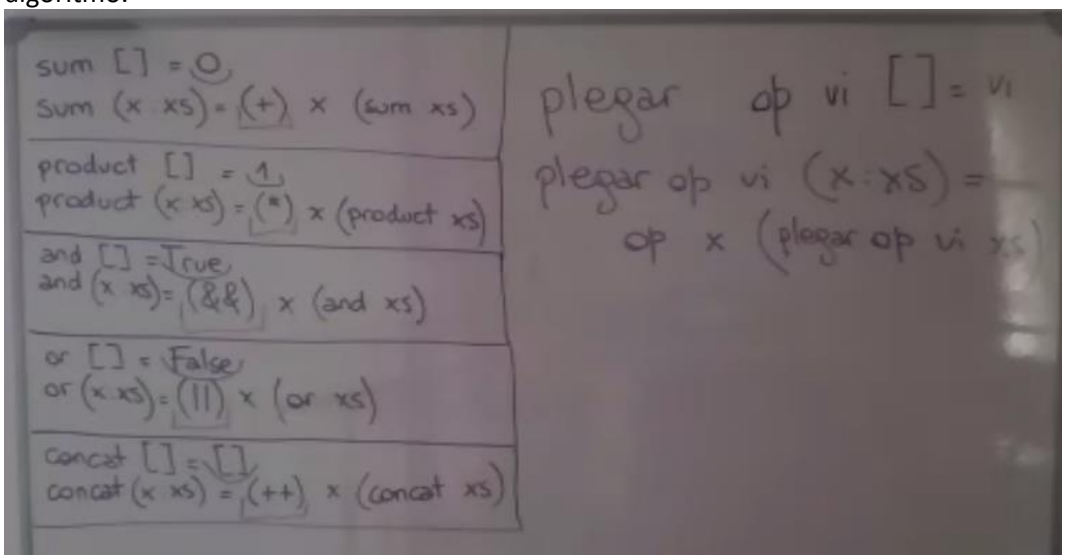
Establecemos los casos base

Definición recursiva

length - -> toma una lista y nos devuelve la cantidad de elementos que hay en esa lista.

FOLD / PLEGAR

Todas estas funciones tienen en común que reciben una lista, se llaman recursivamente y varían en el operador y el caso base. Por ello, se realiza una función genérica que resuelva este algoritmo.



IDAÑEZ, LUCIA MARÍA

A esta función le pasamos la lista, el operador que queremos usar y el "valor inicial" en caso de que la lista esté vacía. (Está definida de forma prefija)

TIPO: (a -> b -> b) -> b -> [a] -> b

operador | Valor inicial | Lista

CASO BASE

plegar operador valorInicial [ø] = valorInicial

En caso de que sea la lista vacía, **plegar** tiene que devolver el "valor inicial" (que va a ser tipo "b")

CASO RECURSIVO

plegar operador valorInicial (X : XS) = (operador) X (plegar operador valorInicial XS)

Recibe el operador, valor inicial y recibe una lista de al menos un elemento.

Aplica la **operación** a la **cabeza de la lista** y al **resultado de llamar recursivamente a la función que estoy definiendo** (fold/plegar) que recibe operador, el valor inicial y xs (cola de la lista)

Las listas son "plegables".
Toda esta función **plegar** está resumida en la función **fold**. "**fold**" procesa una estructura de datos mediante una operación para construir un valor.
La idea de fold se basa en reducir un conjunto a un solo resultado.

Listas en No-Tan-Alto Nivel
fold :: (a -> b -> b) -> b -> [a] -> b
(a -> b -> b) → Función reductora
b → semilla: resultado en caso de que el conjunto esté vacío

Listas en Bajo Nivel
Forma de tipar lista - -> (x : xs) siendo x la cabeza y xs la cola
head → la aplico y me devuelve el primer elemento de la lista
tail → me devuelve la lista menos la cabeza de la lista
last → toma el último elemento de la lista
null→ nos dice si la lista está vacía
elem → nos indica si un elemento forma parte de una lista (elem elemento lista)

Listas en Alto Nivel
all :: (a -> Bool) -> [a] -> Bool → Evalúa si todos los elementos de la lista cumple con esa condición. Esa condición puede ser una composición de funciones.

```
all :: (a -> Bool) -> [a] -> Bool
"Cuantificador universal"
```

any :: (a -> Bool) -> [a] -> Bool → Evalúa si al menos uno de esos elementos cumplen con esa condición. Esa condición puede ser una composición de funciones.

```
any :: (a -> Bool) -> [a] -> Bool
"Cuantificador existencial"
```

IDAÑEZ, LUCIA MARÍA

Clase 11/5

Definimos una función doble x = x + x, tenemos la siguiente función "doble (doble 3)".
Una forma de evaluar esta función es la siguiente:

> doble (doble 3)
> doble (3 + 3)
> doble (6)
> 6 + 6
> 12

CALL BY VALUE → evalúa primero lo que se pasa por parámetro

Esta forma es llamada **call by value**, ya que **primero evalúa el valor, es decir el parámetro**.

Existe otra forma de evaluar llamada **call by name** la cual **resuelve primero lo de más afuera**.
En este caso sería la función "doble", de la siguiente manera:

> doble (doble 3)
> (doble 3) + (doble 3) <- - resuelvo la función que está más afuera, en este caso la suma
> (3 + 3) + (doble 3)
> 6 + (3 + 3)
> 6 + 6
> 12

CALL BY VALUE → resuelve primero lo de más afuera

Entre ambas formas de evaluar podemos notar que ambas llegan al mismo resultado, esto quiere decir que no importa el método de llamar a la función tengamos, siempre llegamos al mismo resultado. Sin embargo, existen algunas ventajas en **call by name**, ejemplo:
->Tenemos la función **fst** :

fst (1, 2) = 1 <- - sin importar el método de evaluación, nos da 1
fst (1, 1/0) = 1 <- - Este caso, si lo evaluamos por **call by value** nos va a resolver todo el parámetro, esto quiere decir que va a intentar resolver la operación (1/0) y es posible que "rompa". En el caso de **call by name**, no evalúa todo el parámetro, solo lo que necesita la función. Evalúa solo lo que tiene que evaluar (lo mismo que nos devuelve).

La utilización de ambas formas de evaluar varía de acuerdo con el caso, es decir depende el contexto de nuestras necesidades utilizar cada método.
(undefined) → *hace que Haskell tire un error*

La forma de evaluar **call by name** cada vez que resuelve una expresión que es igual, la resuelve para todos al mismo tiempo, es decir:

>doble (doble 3)
> (doble 3) + (doble 3) < - - expresiones iguales
> (3 + 3) + (3 + 3) < - - expresiones iguales
> 6 + 6
> 12

Shared Memory

FORMA DE EVALUAR DE HASKELL: CALL BY NAME Y SHARED MEMORY

Esto es conocido como **shared memory** y **call by name**, ambas son la forma en la que Haskell y otros lenguajes evalúan. Esta forma se llama **LAZY EVALUATION (Evaluación perezosa)**.
Call by value es conocido como **EAGER EVALUATION (Evaluación ansiosa)**.
La **evaluación ansiosa** la utilizan casi todos los lenguajes de programación.

No siempre se puede hacer lazy evaluation, esta misma se acompaña de que no hay "efecto". En Haskell se puede hacer la evaluación por call by name ya que evaluar una expresión antes o después da el mismo resultado, es decir no se cambia el valor de la expresión y/o variables, debido a que en Haskell no hay variables sino valores. La idea del lazy evaluation va arraigada a la idea de que "no podemos cambiar el mundo".

“undefined” es una función que calza en cualquier lado. Al momento de colocarlo en una lista, tupla o lo que sea y mostrarlo, falla. Sin embargo, trabaja de manera **lazy** ya que antes de llegar al valor que falla muestra todos los valores anteriores. Esto nos da el poder de trabajar con listas potencialmente infinitas ([1..] lista infinita).
Ejemplo:

take 5 [1..] < - - nos da los primeros 5 elementos de esa lista infinita

También las listas se pueden escribir por rango, es decir podemos poner

[1..10] <-- es la lista del 1 al 10.

No en todas las listas se podrán hacer estas operaciones, en algunos casos la función quedará “colgada”.
Ejemplos:

all even [2, 4, 6..] < - - este caso se queda colgado, ya que, por más que semánticamente entendamos que es una lista de infinitos números pares, Haskell evaluará todos los casos hasta el infinito.

También podemos hacer estas listas con letras [‘a’..’z’] < - - es una lista con las letras de la ‘a’ a la ‘z’. Esto mismo con strings no se puede hacer ya que no son numerables.
[1..] → Esto mismo se puede hacer con la función enumFrom num y nos hace una lista infinita de ese número (enumFrom - -> “enumerar desde”).
[1 .. 10] →Esto se resuelve con la función enumFromTo num1 num2 y nos hace la lista desde el num1 hasta num2 (enumFromTo - -> “enumerar desde - hasta”).

Si queremos una constante recursiva de muchos unos (1), hacemos:

```
unos :: [Int]
unos = 1 : unos < - - compone ese 1 en una lista con otros 1
```

En esta constante, aunque es recursiva, no tenemos un caso base y a diferencia de las funciones recursivas, ésta no converge sino que diverge. Esto quiere decir que podemos hacer funciones que, en vez de tomar una lista y llegar a un valor, toman un valor y nos generan una lista, en este caso una lista infinita.

Así como la familia de funciones fold toman una lista y convergen a un valor, existe la familia de funciones **unfold** que toman un valor y generan una lista.
Tenemos la función **<u>iterate</u>**:
iterate f valor = valor : iterate (f valor)
cycle [a] [s]= [as, as, as, as…] →solo con listas, no números
repeat valor = [valor..] → recibe un valor y genera la lista infinita de ese valor
replicate n valor = [valor * n] → nos da una lista de “n” veces ese valor repetido

foldr → acumulativo
foldl → tail recursion - -> se llama recursivamente a sí mismo

IDAÑEZ, LUCIA MARÍA 😸

EJERCICIOS (DIS)FUNCIONAL: COSAS QUE NO HAY QUE HACER Y ERRORES

`tieneNombreLargo mascota = length (fst mascota) > 9 == True`
En este caso el "==True" es redundante ya que `length (fst mascota) > 9` genera un bool. En el caso de que queramos que sea VERDADERO cuando es FALSO, no es necesario poner "==False", usaremos el operador lógico **not**.

```
sumarEnergia (Persona _ energia _ _) = (Persona _ (energia + 5) _ _)
```
El guion no puede ir a la derecha del =, porque el guion bajo es un patrón que marca lo que no nos interesa, entonces no puede aparecer a la derecha del = si no nos interesa.
Si tenemos que devolver una `Persona` completa, necesitamos todos los datos de la persona, ejemplo:
```
sumarEnergia (Persona nombre energia edad hobbies) =
                        (Persona nombre (energia + 5) edad hobbies)

Ó sumarEnergia persona = persona{energía = energía + 5 }
```

`triplicarLosPares numeros = (map (*3) . filter . even) números`
El (.) entre **filter** y **even** no debería estar, ya que filter recibe dos parámetros y en la composición solo podemos recibir uno.
La manera correcta es:
```
triplicarLosPares numeros = (map (*3) . filter even) números
```

`sonTodosMamiferos animales = all (==True) (map esMamifero animales)`
Esto es lo mismo que hacer `all esMamifero animales`, ya que `map esMamifero` nos devuelve una lista de booleanos y al preguntar si todos son mamíferos con "==True", es redundante cuando ya tenemos la función "esMamifero".

`sonTodosMamiferos' animales = (and . map esMamifero) animales`
Esto está mal, ya que es lo mismo hacer `all esMamifero animales`. En este caso agarra convierte una lista de animales en booleanos con la función "esMamifero" y con "and" toma esa lista de booleanos y verifica que sean todos True.