IDAÑEZ, LUCIA MARÍA
PARADIGMAS DE PROGRAMACIÓN

Pattern Matching Avanzado:
Explicado con 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 esta construcción.
→Función Identidad

Ya conocemos la función identidad, sin embargo esta vez la realizaremos mediante patrones, de la siguiente manera:

```
id :: Nota -> Nota
id (Nota elValor elDetalle) = Nota elValor elDetalle
```

Observemos que de este lado rompimos con Nota en su construcción

De este lado reconstruimos Nota con sus mismos valores

Otro patrón:

```
id :: Nota -> Nota
id nota@(Nota valor observaciones) = nota
```

La arroba es una herramienta que sirve para darle nombre a todo un patrón, es decir descomponemos lo que recibimos (Nota valor observaciones) y lo llamamos "nota" para usarlo de forma "completa".

Pensamos otra función:
notaFinal → suma entre los parciales / 3

Esta primera opción es mediante el Pattern Matching.

Recibe un solo parámetro "Alumno"

```
notaFinal :: Alumno -> Nota
notaFinal (Alumno _ (Nota fun _) (Nota log _) (Nota obj _)) =
    (fun + log + obj) `div` 3
```

Descomponemos la variable que recibe en la forma que esperamos que tenga, es decir primero descomponemos Alumno, el cual tiene anidado Nota y extraemos los campos que necesitamos

Esta forma está muy arraigada a la forma en que definimos los parámetros.
Como definimos funciones de acceso podemos utilizarlas para estos casos.

```
notaFinal alumno = ( valor (notaFuncional alumno) +
                     valor (notaLogico alumno) +
                     valor (notaObjetos alumno)
                     ) ` div ` 3
```

valor opera sobre **notaFuncional** que este a su vez opera sobre **alumno**
Leer de atrás para adelante, va de general a específico

IDAÑEZ, LUCIA MARÍA

Pensamos otra función:
GUARDAS
aprobado → nos va a decir si el alumno va a estar aprobado teniendo en cuenta el año en que se anotó

```
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 menor a 4.
Entonces, ¿cómo resolvemos esto? Sencillo, con la herramienta **guarda** (|) que está dentro 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
```

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

Lo que está escrito son las condiciones que queríamos poner, es decir la sentencia da True si el alumno cumple con ellas. Entonces del lado derecho está aquello que nos resuelve la función.
Debemos separar bien el cuerpo de la función de la condición para usarlo.

Cada sentencia aplica una guarda, ese | tiene a su derecha una expresión booleana que al ser evaluada da como resultado una expresión booleana.
Con "otherwise" como sentencia, siempre nos da True y nos sirve para cualquier otro caso que querríamos cubrir.
"Subir un punto: dada una nota, incrementar su valor en 1"
Hacemos el código:

```
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)
```

Entonces construimos una función que recibe una Nota y retorna una Nota.

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

Construimos una nueva **Nota** con el constructor **Nota**

Cambia su **valor** que es el resultado de obtener el valor de la nota, para ello llama a la función **valor** que a partir de **nota** le sabe dar su valor y + 1, así construimos la nueva **Nota** junto con el mismo detalle de la otra Nota

IDAÑEZ, LUCIA MARÍA

Esto mismo se puede escribir de la siguiente manera:

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

En lugar de usar el constructor, 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.

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 de 1 parámetro.

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

Formas de crear funciones:

→Definición normal de una función;

→Composición;

→Aplicación parcial;

→Expresiones Lambda

Expresiones Lambda:

Es una función anónima 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

```
Función anónima

type Persona = (String, Int)

cumplirAños :: Persona -> Persona
cumplirAños (nombre, edad) = (nombre, (\unNumero -> unNumero + 1) edad)
```

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:

Las guardas nos permiten definir una función por partes.

A la derecha del | debe ir una condición (es decir, un booleano).

Definición local → se usa dentro de una función mediante “where” y a continuación la función.

IDAÑEZ, LUCIA MARÍA

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

Como está definida de manera local no es necesario pasarle los parámetros, las puede usar directamente.

```haskell
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
```

CLASE 4/5

```haskell
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)
```

Entonces construimos una función que recibe una Nota y retorna una Nota.

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

Construimos una nueva **Nota** con el constructor **Nota**

Cambia su **valor** que es el resultado de obtener el valor de la nota, para ello llama a la función **valor** que a partir de **nota** le sabe dar su valor y + 1, así construimos la nueva **Nota** junto con el mismo detalle de la otra Nota

Esto mismo se puede escribir de la siguiente manera:

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

En lugar de usar el constructor, 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.

IDAÑEZ, LUCIA MARÍA

IDAÑEZ, LUCIA MARÍA

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

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
```

(!!) - - > tomar por índice de la lista, es decir toma el elemento de la lista con el índice que le pasemos.

Algunos ejemplos de **recursividad**:

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

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

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

Establecemos los casos base

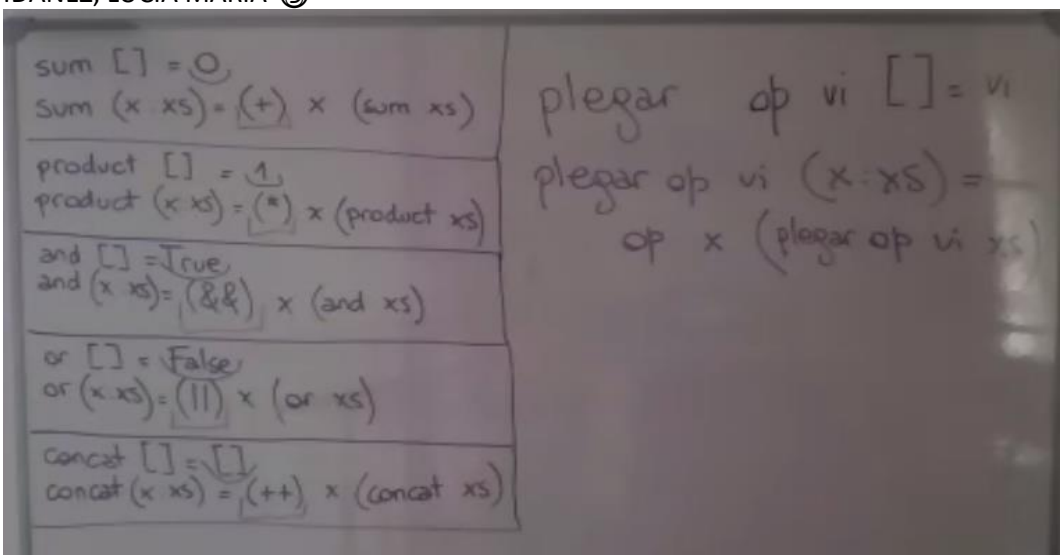
Definición recursiva

Establecemos los casos base

Definición recursiva

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

plegar operador valorInicial [ø] = valorInicial

En caso de que sea la lista vacía, **plegar** tiene que devolver el "valor inicial"

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

Recibe el operador, valor inicial y 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** que recibe operador, valor inicial y xs

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 va ligada a reducir un conjunto a un 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

IDAÑEZ, LUCIA MARÍA

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.
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.

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

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

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

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**.

IDAÑEZ, LUCIA MARÍA 😸
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.
[1 .. 10] →Esto se resuelve con la función enumFromTo num1 num2 y nos hace la lista desde el num1 hasta num2.

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 iterate:
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

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)
```

```
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
```

IDAÑEZ, LUCIA MARÍA

`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.

IDAÑEZ, LUCIA MARÍA

```
abrirVentanas :: Casa -> Casa
prenderEstufa :: Casa -> Casa
encenderElAireA :: Casa -> Int -> Casa
mudarseA :: String -> Casa -> Casa

miCasaInteligente = Casa
   { direccion = "Medrano 951",
     temperatura = 26,
     reguladoresDeTemperatura = [abrirVentanas, prenderEstufa,
         mudarseA, encenderElAireA 24]
   }
```

Podemos tener una lista de funciones siempre que todas sean del mismo tipo.

Como en una lista de funciones todas deben tener el mismo tipo, en este caso no sucede por dos errores, el primero es error de tipos con "mudarseA" ya que le falta el String (dirección) para que sea del tipo Casa->Casa y el segundo caso es "encenderElAireA" ya que le estamos pasando primero un 24 cuando deberíamos pasarle una Casa por lo tanto hay un error ya que 24 no es una Casa, para que funcione "encenderElAireA" deberíamos dar vuelta los parámetros con "flip encenderElAireA".

```
esBeatle _ = False
esBeatle "Ringo" = True
esBeatle "John" = True
esBeatle "George" = True
esBeatle "Paul" = True
```

En este caso los patrones están mal, el caso "otherwise" está al principio cuando debería estar al final, cualquier nombre que entre por esta función dará False. Esto lo podemos arreglar de dos maneras:

1.

```
esBeatle "Ringo" = True
esBeatle "John" = True
esBeatle "George" = True
esBeatle "Paul" = True
esBeatle _ = False
```

2.

```
esBeatle alguien = elem alguien ["Ringo", "John", "George", "Paul"]
```

Si la función devuelve un bool, está prohibido usar guardas.

```
sumaDeLasEdadesRecursiva [] = 0
sumaDeLasEdadesRecursiva lista =
   edad (head lista) + sumaDeLasEdadesRecursiva (drop 1 lista)
```

En este caso hay dos maneras de solucionar los "errores":

1.

```
sumaDeLasEdadesRecursiva [] = 0
sumaDeLasEdadesRecursiva lista =
   edad (head lista) + sumaDeLasEdadesRecursiva (tail lista)
```

2.

```
sumaDeLasEdadesRecursiva [] = 0
sumaDeLasEdadesRecursiva (x : xs)=
   edad x + sumaDeLasEdadesRecursiva xs
```

IDAÑEZ, LUCIA MARÍA

```
abrirVentanas casa = casa { direccion = direccion casa, temperatura =
temperatura casa - 2, reguladoresDeTemperatura = reguladoresDeTemperatura casa
}
```

Este es un error a nivel Haskell, ya que para modificar la temperatura de la casa solo era necesario modificar ese campo en vez de recibir toda la casa. Se soluciona de la siguiente forma:

```
abrirVentanas casa = casa {temperatura = temperatura casa - 2}
```

```
agregarValor valor indice lista =
   take (indice - 1) lista  ++ [valor] ++ drop indice lista
```

Lo que hace esta función es insertar un valor en una lista. Hay una forma de escribir esto de manera más simple:

```
agregarValor valor indice lista =
   take (indice - 1) lista  ++ valor : drop indice lista
```

```
poneleUnNombre numeros = (sum (map (*3) (filter even numeros))) < 100
```

En este caso está mal, ya que se desperdicia el concepto de composición. Se resuelve de la siguiente manera:

```
poneleUnNombre numeros = (<100). sum. map (*3). (filter even)$ numeros
```