Resumen Programación Funcional

Tema 1. Tipos y Clases Prelude

Tipos Primitivos

```
> Bool --True, False
> Char --'a', '3', ...
> Int --Enteros con longitud máxima.
> Integer --Enteros sin longitud máxima.
> Float --Números de coma flotante de precisión simple.
> Double --Números de coma flotante de con doble memeria.
> undefined --Tipo polimórfico para todos los tipos anteriores.
```

Algunas veces es necesario "castear" variables a otro tipo, para ello, utilizamos `<variable>::<tipo>`.

Tipos Compuestos

```
> [t1, t2, ..., tn] --Tipo Lista.
-- Todos los elementos son del mismo tipo.
> (x1, x2, ..., xn) --Tipo Tupla.
-- Cada elemento puede tener su tipo.
> Tipo Funcion:
  1. No currificadas --asociado por la izquierda.
  f: A x B x C -> D => f(x, y, z) = r
  2. Currificadas --Aplicacion parcial asociado por la derecha.
  f: A -> (B -> (C -> D)) => ((f x) y) z = r

> Tipo polimórmifico.
-- Empiezan por minúscula, son variables de tipo.
> Tipo dobrecargado.
-- Restricción de clase.
```

Clases Básicas.

Una **clase** es una colección de tipos con operaciones (métodos).

```
> Eq          -- Tipos cuyos elementos pueden compararse.
-- Métodos como (==), (/=).
> Ord         -- Heredan de Eq y se pueden ordenar.
-- Métodos como (<), (>), (<=), (>=), min, max.
```

```
> Show        -- Tipos que se pueden convertir en String o Char.
-- Método show : a -> String.
> Read        -- Contiene las mismas instancias que Show y tipos legibles.
-- Método read: String -> a.
> Num         -- Tipos numéricos.
-- Métodos como --(+), (-), (*), negate, abs, signum.
> Integral    --Números enteros (Int, Integer).
-- Métodos como --div, mod.
> Fractional --Números fraccionarios (Float, Double).
-- Métodos como --(/), recip
```

Podemos crear nuestros propios operadores y cambiarle la prioridad a los operadores:

```
> infixr <prioridad> <operador> --Asociación por la derecha.
> infixl <prioridad> <operador> --Asociación por la izquierda.
```

Los operadores pueden utilizarse de forma infija o prefija, es decir.

$$a \oplus b = (\oplus)ab = (\oplus a)b$$

Tema 2. Evaluación de Expresiones

Nunca produce un **efecto colateral** [No modifica el estado de su entorno], es decir el resultado de evaluar una expresión es independiente del contexto.

```
> [[e]] = v .
--Ejemplo
[[3 + 1]] = 4, [[let x = 2 in x * 3]] = 6
> [[e1, e2]] = v, tales que
e1:T -> T', e2:T => (e1 e2):T' => v en T'
```

Funciones currificadas

```
f:T1 -> T2 -> ... -> Tn -> T
f e1 e2 ... en = (...(f e1) e2)...en)
[[f e1]]  --T2 -> .. -> Tn -> T
[[f e1 ... en-1 en]] esta en T

-- La flecha de los tipos (->) asocia por la derecha.
-- Ejemplo.
> A -> B -> C -> D = A -> (B -> (C -> D))
                   = A -> (B -> C -> D)
                  != (A -> B) -> C -> D
                  != A -> (B -> C) -> D
-- La aplicación en expresiones asocia por la izquierda
-- Ejemplo.
> f x y z = ((f x) y) z
          = (f x y) z
         != f (x y z)
         != f (x y) z
```

NOTA: Una funcion puede ser **argumento** o **resultado** de otra. Ejemplo de equivalencia de tipos (¿t1 = t2?).

```
t1 = (A -> B -> A) -> A -> (B -> C)
t2 = (A -> (B -> A)) -> (A -> (B -> C))

t1 = (A -> B -> A) -> A -> B -> C
t2 = (A -> B -> A) -> (A -> B -> C)
t2 = (A -> B -> A) -> A -> B -> C = t1
-- => El tipo de t1 es equivalente a t2.
```

Truco: Englobar la expresión entre paréntesis y aplicar recursivamente el **quitar paréntesis si a la derecha hay un paréntesis**.

Funciones con guardas

```
f:T1 -> ... -> Tn -> T
f p1 ... pn = e1 (p1 .. pn son patrones lineales)
...
f p'1 ... p'n (p'1 ... p'n son patrones)
    | b1        = ex1
    | b2        = ex2
    ...
    | bk        = exk
    | otherwise = ex
    where {v1=<operaciones>; ...; vj=<operaciones>}
```

Ajuste de una expresión a un patrón

Los patrones p pueden ser:

1. x sobre una expresión e => Sustitución [x / e]
2. _ sobre una expresión e => Ninguna sustitución
3. C sobre una expresión e => Ninguna sustitución
4. $p_1..p_n$ sobre expresiones $e_1..e_n$ => Si e_i se ajusta con p_i, Se reunen las sustituciones de los n ajustes.
5. C $p_1..p_n$ sobre expresiones C e1..en => Si e_i se ajusta con p_i, se reunen las sustituciones de los n ajustes.

Valor indefinido

> Cuando una expresion no puede evaluarse porque da un **error de ejecución** o es **infinita**, se dice que está **indefinida** (bottom) y utilizamos la simbología $\perp$ para definirlo.

```
> [[e]] = bottom
-- Ejemplos:
> [[ [1+1, div 1 0] ]] = [2, $\bot$]
> [[ undefined ]] = $\bot$
> [[ [1]++head [] ]]=1:$\bot$
```

Funciones Estrictas

```
> f:T -> T'        es estricta <=> f bottom = bottom
> f:T1 -> Tn -> T  es estricta en el i-ésimo argumento
-- Sí y solo sí, f e1 ... ei-1 bottom ei+1 ... en = bottom
```

Composición de funciones

Definimos la composición de funciones como f(g(x))

```
> (.)::(b -> c) -> (a -> b) -> (a -> c)
  (.) f g x = f (g x)
-- Ejemplo.
  (head . tail) ls = (head (tail ls))
  ((^ 2) . (3 *)) 2 = 2 ^ (3 * 2)
```

Tema 3. Funciones Generales

En esta lección hay que tener en cuenta la **evaluación perezosa**, es decir, si una condición se cumple sin tener que evaluar el resto, no se evaluará.

El **compilador** va a detectar que los tipos se cumplen estrictamente (*Error de tipos*), luego en la ejecución se comprueba que la expresión tiene valor correcto (*Error de ejecución*).

Funciones para las tuplas

```
> fst --devuelve el primer elemento de la tupla.
Def. fst::(a, b) -> a.

> snd --devuelve el segundo elemento de la tupla.
Def. snd::(a, b) -> b.
```

Funciones para las listas

Creación de listas

Las listas puedes tener un tamaño finito, pero estas pueden tener tamaño infinito si éstos son de tipo numérico.

```
> [Int, Int..Int].            Tamaño finito
> [Int, Int, ..., Int, ..].   Tamaño infinito
> [<operacion con a> | a<-A]. Tamaño dependiente de A
> take n (iterate f k).       Tamaño dependiente de n
```

Para iterate, **k el primer numero de la lista** y **f es la expresión que evalua** al término anterior para generar el siguiente.

Funciones básicas

```
> head --devuelve el primer elemento de la lista.
Def. head::[a] -> a

> tail --devuelve una lista sin la cabeza.
Def. tail::[a] -> [a]

> last --Último elemento de la lista.
Def. last::[a] -> a

> (!!) --Selección n-simo elemento.
Def. (!!)::[a] -> Int -> a
```

```
> (:) --Añade un elemento a la lista.
Def. (:)::a -> [a] -> [a]
```

Funciones atributivas

```
> null --Devuelve True o False si está vacía o no la lista.
Def. null::[a] -> Bool

> length --Devuelve el tamaño de la lista.
Def. length::[a] -> Int

> (++) --Concatena dos listas-
Def. (++)::[a] -> [a] -> [a]

> elem --Test de pertenencia a una lista.
Def. Eq a => a -> [a] -> Bool

> all -- Comprueba que todos los elementos cumplen una función.
Def. all::(a -> Bool) -> [a] -> Bool

> any -- Comprueba que alguno de los elementos cumple una función.
Def. any::(a -> Bool) -> [a] -> Bool
```

Funciones de manipulación total.

```
> init --Todos los elementos menos el último.
Def. init::[a] -> [a]

> reverse --Inversión de una lista.
Def. reverse::[a] -> [a]

> concat --Desenrolla una lista de lista en una lista.
Def. concat::[[a]] -> [a]

> take --Devuelve una lista de los n primeros elementos.
Def. take::Int->[a] -> [a]

> takeWhile --Toma los primeros (max n) elementos que cumplan f
Def. takeWhile::(a -> Bool) -> [a] -> [a]

> dropWhile --Quita los primeros (max n) elementos que cumplan f
Def. dropWhile::(a -> Bool) -> [a] -> [a]

-- Nota: takeWhile y dropWhile acaban cuando f no se cumple.
```

```
> span --Devuelve un par con elementos de takeWhile y dropWhile.
Def. span::(a -> Bool) -> [a] -> ([a], [a])

> break --Es span, aplicando la negación del filtro.
Def. break::(a -> Bool) -> [a] -> ([a], [a])

> drop --Devuelve una lista sin los primeros n elementos.
Def. drop::Int -> [a] -> [a]

> splitAt --Separa los primeros n elementos de los demás.
Def. splitAt::Int->[a] -> ([a],[a])

> zip --Empareja dos listas elemento a elemento.
Def. zip::[a] -> [b] -> [(a, b)].
-- Nota. Si el tamaño de uno de los dos conjuntos es menor que el otro,
-- entonces se emparejará todos los del conjunto menor.

> zipWidth --Aplica una funcion a dos listas, elemento a elemento.
Def. zipWidth::(a -> b -> c) -> [a] -> [b] -> [c]

> unzip --Desempareja una lista de parejas.
Def. zip::[(a,b)] -> ([a], [b]).

> map --Aplica una función (f) a cada uno de los elementos.
Def. map::(a -> b) -> [a] -> [b]
-- Ejemplos.
> map ($\oplus$ 2) [1, 2] = [2$\oplus$1, 2$\oplus$2]
> map (2 $\oplus$) [1, 2] = [1$\oplus$2, 2$\oplus$2]

> filter --Filtra la lista dada una función
Def. filter::(a -> Bool) -> [a] -> [a]

> foldr --Aplica de forma recursiva una expresión por la derecha.
Def. foldr::(a -> b -> b) -> b -> [a] -> b
-- Ejemplo.
> foldr f e [x1,...,xn] = f x1 (f x2(..(f xn e)..))
-- Si f es un operador $\oplus$
> foldr $\oplus$ e [x1,...,xn] = x1 $\oplus$ ... $\oplus$ xn $\oplus$ e

> foldl --Aplica de forma recursiva la expresión por la izquierda.
Def. foldl::(a -> b -> a) -> a -> [b] -> a
> foldl f e [x1,...,xn] = (f(...(f (f e x1) x2)...) xn
-- Si f es un operador $\oplus$
> foldl $\oplus$ e [x1,...,xn] = e $\oplus$ x1 $\oplus$ ... $\oplus$ xn
```

Funciones matemáticas

```
> sum/product -- $\Sigma$/$\Pi$ de todos los elementos de la lista.
Def. (sum/product)::Num a => [a] -> a

> and/or --Realiza la $\wedge$/$\vee$ de todos los elementos.
Def. and/or::[Bool] -> Bool.
```

Funciones para las funciones

```
> flip --Cambia el orden de los argumentos.
Def. flip::(a -> b -> c) -> (b -> a -> c)

> curry --Currifica una función sobre parejas.
Def. curry::((a,b) -> c) -> (a -> b -> c)
-- Ejemplo.
curry f x y = f (x, y)

> uncurry --Descurrifica una función sobre parejas.
Def. curry::(a -> b -> c) -> ((a,b) -> c)
-- Ejemplo.
curry f (x, y) = f x y

> ($) --Es el operador paréntesis, asegura precedencia.
Def. ($)::(a-> b) -> a -> b

> id --Es la función identidad.
Def. id::a -> a

> const --Es la función constante.
Def. const::a -> b -> a
```

Ajuste de patrones (Pattern Matching)

Haskell hace match con la expresión y sabe cuál es el orden de los elementos, es muy util para listas y tuplas.

```
> Listas.
Pattern Matching (a1:[a2,.., an]) = (x:xs)
    - f (x:xs) = xs <=> tail [a1, a2,.., an] = [a2,.., an]
    - f (x:xs) = x  <=> head [a1, a2,.., an] = a1
> Tuplas.
Patter Matching (a, b) = (x, _)
    - f (x, _) = x  <=> fst (a, b) = a
```

Tema 4. Expresiones condicionales, λ, y listas intensionales.

Expresiones condicionales

```
> if b then e else e'
```

Dónde b es una expresión booleana, e y e' son expresiones del **mismo tipo**, esta función tiene aridad 3.

```
> let {x1 = e1; ...; xn = en} in e'
```

x es una **variable ligada**, ***ei*** la **ligadura** de xi y e' una **expresión principal**, el valor del resultado es **igual** al que **produce la expresión principal**.

```
let x=1:2:x in take 3 x
```

> **NOTA**: No genera errores, su **evaluacion es recursiva**, es decir genera g(x)=1:2:g(x), pero al ser **evaluación perezosa**, en la siguiente iteración g(x)=1:2:(1:2:g(x)), se tomará el valor [1,2,1] (3 primeros de dicha secuencia).

Expresiones λ

Son expresiones que se declaran en línea.

```
> \x -> e
> \x y -> x + y  == \x -> (\y -> e)
> \(x:xs) -> x -- Acepta ajuste de patrones.
> \x y -> case e of
            t1 -> e1
            ...
            t2 -> en
```

Listas intensionales

> Se utilizan para crear listas de expresiones y filtradas, muy útiles a la hora de ahorrar código.

```
> [ e | x1 <- A1, f1 x1 ..., xi <- Ai, fi xi ]
-- Ejemplos utilizando listas y map, filter y concat.
> [x | x <- [1..9], filtro] = map(\x -> x) (filter filtro [1..9])
```

> **NOTA**: El orden de xi influye, ya que se coge de privote el primer elemento de x_1 y se combina con los demás x_n.

Tema 5. Tipos construidos

```
> data T a1..an = C1 t11..t(1(k1)) | ... | Cn tm1..t(m(km))
> C1..Cn::T
> Si Ci = T (En nombre), este es el constructor.
-- Ejemplo 1.
> data Nat = Cero | Suc Nat
> Cero::Nat y Suc::Nat->Nat
-- Ejemplo 2.
> data List a = Nil | Cons a (List a)
> Nil::List a y Cons::a -> List a -> List a
```

> Donde T es un identificador de **constructor de tipos** Ci de **constructor de datos** por lo que no puede haber Ci=Cj con j!=i, a1..an son variables de tipo (puede no tener, n=0).

Tipos monomórficos.

```
> T ::=     TP           // Tipo Primitivo
        |   (T1,..., Tn) // Producto de tipos
        |   [T]          // Lista de elementos
        |   T -> T'      // Tipo Funcional
```

Podemos crear **Alias** que nos van a ayudar a crear Tipos simples (por ejemplo de tipos primitivos).

```
> type <nombre> <valor> = <expresión>
-- Ejemplo.
> type Coordenada = (Float, Float)
> distancia::Coordenada->Coordenada->Float
> distancia (x, y) (a, b) = sqrt (((x-a)^2) + (y-b)^2)
> type Terna a = (a, a, a)
```

A diferencia de la construcción de tipos, **no puede ser recursivo** `type T = (Int, [T])` daría un **error**.

Tema 6. Clases e Instancias

```
> class <nombre> <elemento> where
    <operador1/metodo1>::<tipos1>
    ...
    <operadorm/metodom>::<tiposm>
--Ejemplo.
> class Eq a where
    (==),(/=) : --a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Ahora solo basta instanciar (==) o (/=), por lo que, vamos a declaramos las instancias de la clase.

```
> instance <nombre> <tipo> where
    <operador1> == <operador2> = <valor1>
    ...
    <operadorn> == <operadorn> = <valorn>
--Ejemplo.
> instance Eq Bool where
    False == False = True
    False == True  = False
    True  == False = False
    True  == True  = True
```

> Para ahorrar tiempo a la hora de instanciar otra vez los operadores básicos, podemos extender los métodos a nuestros tipos declarados `data T = ... deriving <Clase Básica>`, es importante ver que **SOLO** se puede extener a las **Clases Básicas**.

Tema 7. Subclases

Podemos querer tener clases que tengan las mismas funcionalidades que el padre, pero además que contengan más propiedades.

```
class <padre> <padre_tipo> => <hijo> <hijo_tipo2> where
    <definiciones>
-- Ejemplo.
> data Ordering LT | EQ | GT
> class Eq a => Ord a where
    infix 4 <, 4 ,<=,>=
    compare::a -> a -> Ordering
    (<), (<=), (>=), (>)::a -> a -> Bool
    max, min::a -> a -> a
    compare x y | x == y = EQ
                | x <= y = LT
                | otherwise = GT
    x <= y = compare x y /= GT
    x < y  = compare x y == LT
    ...
```

Nota 1: Todos los **tipos primitivos, listas y tuplas tienen orden**, es decir, la clase Ord tiene instanciada todos los tipos primitivos.
Nota 2: Si tenemos un tipo de dato T (data T =...) y queremos evaluar el orden de dos elementos x e y de T. \1. Se compara las constructoras más externas de x e y que aparecen en la definición de T. \2. Si son iguales, se comparan las tuplas de argumentos.

Tema 8. Entrada y Salida

Esto supone un problema para Haskell ya que se supone que los estados no cambian por lo que siempre se debería leer la misma entrada por lo que se crea la constructora de tipos `IO` que pertenece a `Monad` (También `Maybe` o `[]`.

```
> getInt::IO Int                          -- Lee un entero
> getChar::IO Char                        -- Lee un carácter.
> getLine::IO String                      -- Lee una cadena.
> putStr::String -> IO ()                 --
> putChar::Char -> IO ()                  -- Escribe un carácter.
> putStr::String -> IO ()                 -- Escribe una cadena.
> print : --Show a => a -> IO ()          -- Escribe un info de un tipo.
> return::a -> IO a                       --
> (>>=): --IO a -> (a -> IO b) -> IO b    --
> do x <- e  =|=> e >>= \x -> e'          --
     e'      =|
```

Nota: () es un 0-upla, que tiene el valor de (). las acciones de tipo `IO ()` no generan valor as +ociado.

Resumen Programación Lógica

Tema 1. Hechos y Reglas

Un programa lógico está formado por cláusulas que pueden ser de dos tipos:

1. **Hechos**: `H :- true` (Se puede omitir true, quedando H)
2. **Reglas**: `H :- B` (H es cierto si se cumple B).

```
> <nombre>(<arg1>,..,<argn>) :- <regla1>, ..., <reglam>.
% Ejemplo
> padre(X, Y) :- progenitor(X, Y), hombre(X).
```

> **NOTA**: Prolog permite tener el mismo nombre para dos predicados de distinta aridad.

Ejecutar programas implica realizar "preguntas" a los objetivos

```
?- <nombre>(<valor1>,..,<valor2>)
% Ejemplo
?- padre(pedro, maria)
```

Podemos tener objetivos compuestos y se interpretan como la **conjunción** ($\vee$) de sus literales

```
?- <nombre1>(<valor11>,..,<valor1j1>), ..., <nombre2>(<valor1jm>,..,<valor2jk>)
```

La evaluación de un objetivo puede devolver dos resultados lógicos:

1. **Si**: Cuando el objetivo es consecuencia de las reglas del programa.
2. **No**: Caso contrario. ## Tema 2. Datos básicos ### Variables, constantes y estructuras.

```
> _<nombre> o <Nombre>           % Son variables
> <nombre>                       % son constantes
> <nombre>(<valor1>,...<valorn>) % Estructuras, functores
% Ejemplos
> _x o X                                                % Variable (Aridad 0)
> v                                                     % Constante (Aridad 0)
> libro(titulo(don_quijote), autor(miguel_cervantes)) % Estructura (Aridad 2)
% Notación infija, prefija y postfija:
> es_un(fluffy, gato) == fluffy es_un gato
```

Tema 3. Unificación

> Es el mecanismo utilizado que se utiliza para **pasar valor** y **devolver resultados**, si una de las ramas da afirmativa, es el resultado que se

devuelve. ### Estrategias de unificación

1. **Sustitución**:
    1. Holi
2. Hola

Trucos para Examen.

Sea f definida por f ... . El tipo de *f* es:

1. Fijarse en las posibles opciones.
2. Simplificar, quitar paréntesis utilizando la regla expuesta en el **Tema 2**, *Funciones currificadas*.

Sea f definida por las siguientes ecuaciones: f a11 .. a1k = s1 ... . f ank .. ank = sn f a1 ... ak =