# El lenguaje Haskell

**HASKELL ES UN LENGUAJE DE PROGRAMACIÓN. ES UN LENGUAJE DE TIPOS POLIMÓRFICOS, DE EVALUACIÓN PEREZOSA, PURAMENTE FUNCIONAL, MUY DIFERENTE DE LA MAYORÍA DE LOS OTROS LENGUAJES DE PROGRAMACIÓN. EL NOMBRE DEL LENGUAJE SE DEBE A HASKELL BROOKS CURRY. HASKELL SE BASA EN EL LAMBDA CÁLCULO.**

# Haskell

- Haskell es un lenguaje de programación moderno, estándar, no estricto, puramente funcional. Posee todas las características avanzadas, incluyendo polimorfismo de tipos, evaluación perezosa y funciones de alto orden. También es un tipo de sistema que soporta una forma sistemática de sobrecarga y un sistema modular.

- ¿POR QUE USAR HASKELL? Escribir en grandes sistemas software para trabajar es difícil y caro. El mantenimiento de estos sistemas es aún más caro y difícil. Los lenguajes funcionales, como Haskell pueden hacer esto de manera más barata y más fácil. Haskell, un lenguaje puramente funcional ofrece:

1. **Un incremento sustancial de la productividad de los programas.**
2. **Código más claro y más corto y con un mantenimiento mejor.**
3. **Tiempos de computación más cortos.**

# Haskell – Cia que utilizan actualmente Haskell

Aquí te compartimos algunas organizaciones que utilizan este lenguaje y de qué manera lo hacen:

- Facebook usa Haskell para combatir el spam. Los ingenieros de Facebook han elegido a Haskell por su rendimiento, soporte de desarrollo interactivo y otras características que hacen de Haskell la mejor opción para su proyecto Sigma.
- NVIDIA utiliza Haskell para el desarrollo de backend de sus GPU.
- Microsoft usa Haskell en su proyecto Bond. Bond es un marco multiplataforma para trabajar con datos esquematizados. Este marco se usa ampliamente en servicios de alta escala.
- J.P. Morgan, el banco más grande de los Estados Unidos, tiene un grupo Haskell en su equipo de Desarrollo de Nuevos Productos.
- Barclays utiliza Haskell por sus exóticas herramientas comerciales.
- IBM, AT&T y Bank of America también utilizan Haskell y soluciones de programación funcional para sus proyectos.

# Ventajas de usar Haskell

Como lenguaje de programación funcional, Haskell tiene beneficios como tiempo de desarrollo más corto, código más limpio y alta confiabilidad. El control estricto de los efectos secundarios también elimina muchas interacciones imprevistas dentro de una base de código. Estas características son especialmente interesantes para las empresas que deben crear software con altas tolerancias a fallas, por ejemplo, industrias de defensa, finanzas, telecomunicaciones y aeroespacial.

# Evalución Perezosa

En programación la evaluación perezosa, o llamada por necesidad, o por su nombre en inglés: lazy evaluation. Es una técnica de evaluación, sí porque hay varias técnicas, que consiste en retrasar el cálculo (o ejecución) de una instrucción hasta que en realidad es necesaria.

Ejemplo:

Evaluación mediante paso de parámetros por valor (o por más internos):

mult (1+2,2+3) = mult (3,5) [por def. de +] = 3*5 [por def. de mult] = 15 [por def. de *]

EVALUACIÓN PEREZOSA

Evaluación mediante paso de parámetros por nombre (o por más externos):

mult (1+2,2+3) = (1+2)*(3+5) [por def. de mult] = 3*5 [por def. de +]

# Funciones de orden superior

## Funciones de orden superior

- Una función es *de orden superior* si toma una función como argumento o devuelve una función como resultado.
- `(dosVeces f x)` es el resultado de aplicar `f` a `f x`. Por ejemplo,

```
dosVeces (*3) 2           == 18
dosVeces reverse [2,5,7]  == [2,5,7]
```

```
dosVeces :: (a -> a) -> a -> a
dosVeces f x = f (f x)
```

# Haskell

- Los programas funcionales son también relativamente fáciles de mantener porque el código es más corto, más claro y el control riguroso de los efectos laterales elimina gran cantidad de interacciones imprevistas.
- Haskell es un lenguaje de programación fuertemente tipado.
- Tipos: Una propiedad importante del Haskell es que es posible asociar un único tipo a toda expresión bien formada. Esta propiedad hace que el Haskell sea un lenguaje fuertemente tipado. Como consecuencia, cualquier expresión a la que no se le pueda asociar un tipo es rechazada como incorrecta antes de la evaluación. Por ejemplo: f x = 'A' g x = x + f x
- La expresión 'A' denota el caracter A. Para cualquier valor de x, el valor de f x es igual al caracter 'A', por tanto es de tipo Char. Puesto que el (+) es la operación suma entre números, la parte derecha de la definición de g no está bien formada, ya que no es posible aplicar (+) sobre un caracter. El análisis de los escritos puede dividirse en dos fases: Análisis sintáctico, para chequear la corrección sintáctica de las expresiones y análisis de tipo, para chequear que todas las expresiones tienen un tipo correcto.

# Haskell

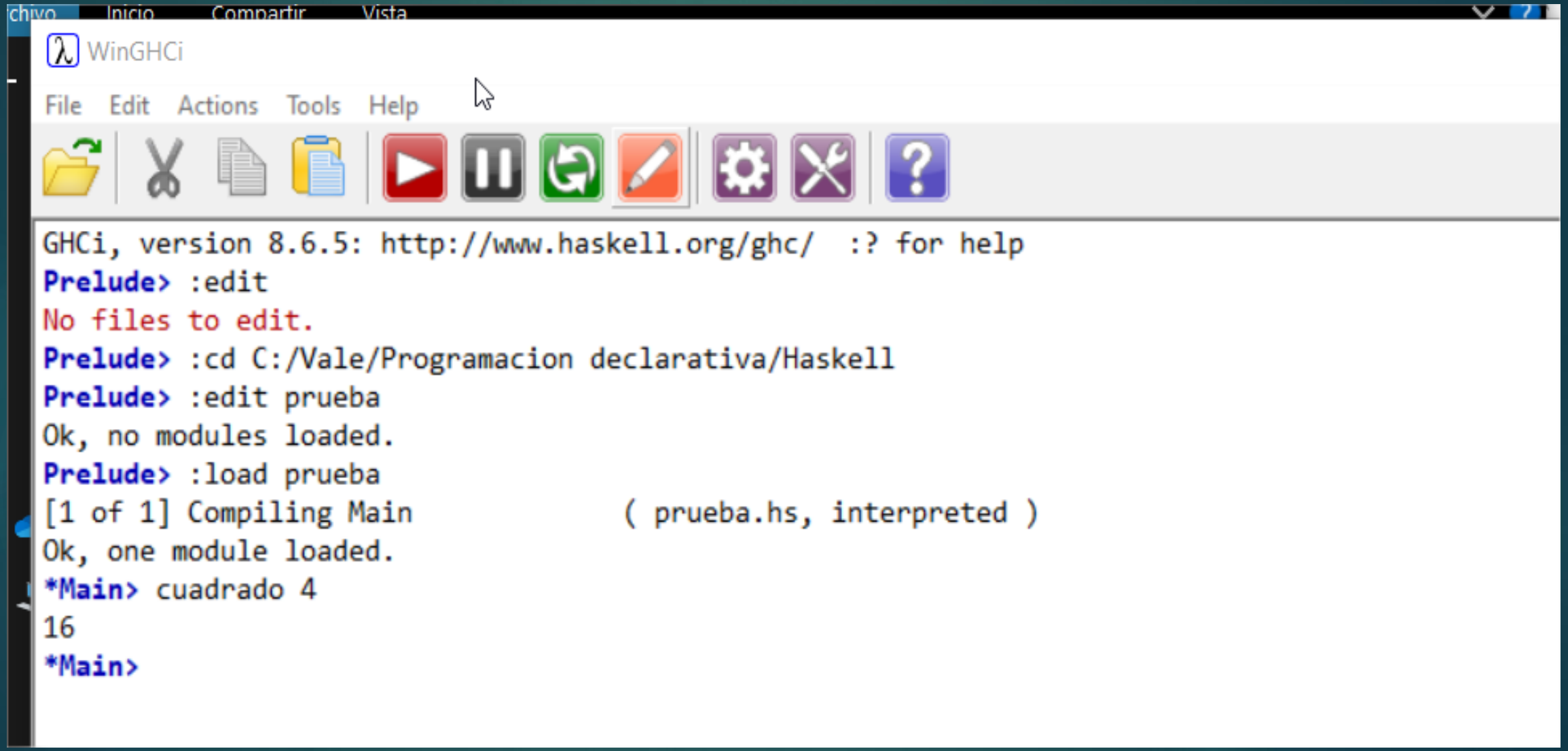
- `:l <arch>` o `:load <arch>` carga el archivo y lo interpreta
- `:q`, `:quit` o `Ctl+D` para salir
- No se pueden usar instrucciones multilínea directamente. Se pueden escribir
  - separadas por punto y coma

```
signo :: (Integral a) => a -> a; signo x = mod x
```

  - encerradas entre llaves

```
:{
signo :: (Integral a) => a -> a
signo x = mod x 2
:}
```

# Haskell


chivo
Inicio
Compartir
Vista
WinGHCi
File
Edit
Actions
Tools
Help
GHCi, version 8.6.5: http://www.haskell.org/ghc/ :? for help
Prelude> :edit
No files to edit.
Prelude> :cd C:/Vale/Programacion declarativa/Haskell
Prelude> :edit prueba
Ok, no modules loaded.
Prelude> :load prueba
[1 of 1] Compiling Main ( prueba.hs, interpreted )
Ok, one module loaded.
*Main> cuadrado 4
16
*Main>

# Tipos de Datos

## Tipos de Datos

- `Num` ⇒ Es un valor numérico
- `Real` ⇒ Es un valor numérico real
- `Fractional` ⇒ Es un valor numérico fraccional
- `Integral` ⇒ Es un valor numérico entero
  - `Int` ⇒ Limitado
  - `Integer` ⇒ Virtualmente infinito
- `Floating` ⇒ Es un valor de punto flotante
  - `Float` ⇒ Precisión simple
  - `Double` ⇒ Precisión doble
- `Bool` ⇒ Es un valor Booleano
- `Char` ⇒ Es un caracter
- `Eq` ⇒ Tiene definida la igualdad
- `Ord` ⇒ Es ordenable
- `Enum` ⇒ Es enumerable
- `Show` ⇒ Se puede mostrar como texto
- `Read` ⇒ Se puede obtener a partir de texto

# Tipos de Datos

## Tipos básicos

- Bool (Valores lógicos):
  - Sus valores son True y False.
- Char (Caracteres):
  - Ejemplos: 'a', 'B', '3', '+'
- String (Cadena de caracteres):
  - Ejemplos: "abc", "1 + 2 = 3"
- Int (Enteros de precisión fija):
  - Enteros entre $-2^{31}$ y $2^{31} - 1$.
  - Ejemplos: 123, -12
- Integer (Enteros de precisión arbitraria):
  - Ejemplos: 126765060022822940149670320537 6.
- Float (Reales de precisión arbitraria):
  - Ejemplos: 1.2, -23.45, 45e-7
- Double (Reales de precisión doble):
  - Ejemplos: 1.2, -23.45, 45e-7

# Comentarios

## Comentarios

Hay dos modos de incluir comentarios en un programa:

- Comentarios de una sola línea: comienzan por dos guiones consecutivos ( $--$ ) y abarcan hasta el final de la línea actual:

```
f :: Integer → Integer
f x = x + 1   -- Esto es un comentario
```

- Comentarios que abarcan más de una línea: Comienzan por los caracteres $\{-$ y acaban con $-\}$. Pueden abarcar varias líneas y anidarse:

```
{- Esto es un comentario
   de más de una línea -}

g :: Integer → Integer
g x = x - 1
```

# Haskell – Operadores Básicos

## Operadores Básicos

- `+` ⇒ Suma
- `-` ⇒ Resta o cambio de signo
- `*` ⇒ Multiplicación
- `/` ⇒ División
- `div` ⇒ División entera
- `mod` ⇒ División modular
- `**` ⇒ Potencia con argumentos `Floating`
- `^` ⇒ Potencia con primer argumento `Num` y segundo `Integral`
- `%` ⇒ Simplifica la relación entre dos `Integral`

# Haskell - Operadores Básicos

## Operadores Básicos

- `==` ⇒ Igual
- `/=` ⇒ Distinto
- `<`, `<=` ⇒ Menor, menor igual
- `>`, `>=` ⇒ Mayor, mayor igual
- `&&` ⇒ Y lógico
- `||` ⇒ O lógico

# Operador Infijo (.)

Dado el ejemplo:

```
impar3 :: Integer -> Bool
impar3 = not . even
```

f . g = \ x -> f (g x)
En el ejemplo hace not (even x)

# Haskell - Tipos de datos

- **Información de tipo** Además de las definiciones de función, en los escritos se puede incluir información de tipo mediante una expresión de la forma A::B para indicar al sistema que "A es de tipo B".

Por ejemplo:

```
cuadrado:: Int -> Int //no es necesario, pero es buena práctica
cuadrado x = x * x
```

- La primera línea indica que la función cuadrado es del tipo "función que toma un entero y devuelve un entero". Aunque no sea obligatorio incluir la información de tipo, sí es una buena práctica, ya que el Haskell chequea que el tipo declarado coincide que el tipo inferido por el sistema a partir de la definición, permitiendo detectar errores de tipos.

# Haskell - Tipos de datos

- Tipos predefinidos Existen varios “tipos” predefinidos del sistema Haskell, éstos se podrían clasificar en: tipos básicos, cuyos valores se toman como primitivos, por ejemplo, Enteros, Flotantes, Caracterses y Booleanos; y tipos compuestos, cuyos valores se construyen utilizando otros tipos, por ejemplo, listas, fu
- Funciones En Haskell las funciones se definen usualmente a través de una colección de ecuaciones.

Por ejemplo, la función inc puede definirse por una única ecuación:

**inc n = n+1**

Una ecuación es un ejemplo de declaración.

Otra forma de declaración es la declaración de tipo de una función o type signature declaration, con la cual podemos dar de forma explícita el tipo de una función; por ejemplo, el tipo de la función

**inc: inc :: Integer -> Integer**

# Haskell

- En el primer ejemplo, el usuario introdujo la expresión

  "(2+3)*8" que fue evaluada por el sistema imprimiendo como resultado el valor "40".

- En el segundo ejemplo, el usuario tecleó "sum [1..10]". La notación [1..10] representa la lista de enteros que van de 1 hasta 10, y sum es una función estándar que devuelve la suma de una lista de enteros. El resultado obtenido por el sistema es: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55

# Haskell – Funciones Básicas predefinidas

x+y  es las suma de x e y

x-y es la resta de x e y

x*y es el producto de x e y

x/y es el cociente de x e y

x^n es x elevado a n, donde n es un número natural

  10^3

x^^n es x elevado a n, donde n es un número entero

 5^^(-2)

x**y es x elevado a y

5**0.25

5**(-1)

16**(1/2)

# Haskell – Funciones Básicas predefinidas

x == y  verifica si x es igual a y

2+3 == 1+4

True

S+3 == 1+0

False

x /= y  verifica si x es distinto a y

2+3 /= 1+0

True

“hola” /= “hola”

false

# Haskell – Funciones Básicas predefinidas

x < y verifica si x es menor a y

2+3 < 7

True

“ayer” < “hoy”

True

x <= y verifica si x es menor o igual a y

2+3 <= 5

True

x > y verifica si x es mayor a y

2+3 > 4

True

x >= y verifica si x es mayor o igual a y

2+3 >= 5

True

# Haskell – Funciones Básicas predefinidas

x && y es la conjunción (and) de x e y

1 < 2+3 && 3+4 <= 9

True

x || y es la disyunción (or) de x e y

1 < 2+3 || 3+7 <= 9

True

x:ys es la lista obtenida añadiendo x al principio de ys

2:[5,3]

[2.5.3]

# Haskell – Funciones Básicas predefinidas

xs ++ ys es la concatenación de xs e ys

[2,4] ++ [3,5,6]

[2,4,3,5,6]

xs !! n es el elemento n-ésimo de xs

[7,9,6,5] !! 2

6

“Sevilla” !! 3

‘i’

abs x es el valor absoluto de x

abs(-7)

7

# Haskell – Funciones Básicas predefinidas

**all p xs** verifica si todos los elementos de xs cumplen la propieda p

all even [2,6,8]

True

**and xs** es la conjunción de la lista de booleanos xs

and [1 < 2+3, 3+4 <= 9, 4 == 7-3]

True

**any p xs** verifica si algún elemento de xs cumple la propiedad p

any even [3,2,5]

true

# Haskell – Funciones Básicas predefinidas

**concat xss** es la concatenación de la lista de listas xss

concat [[2,3],[7],[3,4,5]]

[2,3,7,3,4,5]

**concatMap f xs** aplica la función f a los elementos de xs y concatena el resultado

concatMap (replicate 2) [3,4,5]

[3,3,4,4,5,5]

concatMap (replicate 2) “abc”

“aabbcc”

# Haskell – Funciones Básicas predefinidas

**curry f** es la versión curryficada de la función f

let f (x,y) = x + y

f (2,5)

7

(curry f) 2 5

7

**div x y** es la división entera entre x e y

div 7 2

3

# Haskell – Funciones Básicas predefinidas

**drop n xs** borra los n primeros elementos de xs

drop 2 [7,5,9,6,8]

[9,6,8]

**elem x ys** verifica si x pertenece a ys

elem 3 [5,3,7]

True

**even x** verifica si x es par

even 6

true

# Haskell – Funciones Básicas predefinidas

**filter p xs** es la lista de elementos de la lista xs que verifica el predicado p

filter even [3,4,6,7,5,0]

[4,6,0]

Flter (<6) [3,4,6,7,5,0]

[3,4,5,0]

**fst p** es el primer elemento del par p

fst (5,2)

5

# Haskell – Funciones Básicas predefinidas

**gcd x y** es el máximo común divisor de x e y

gcd 12 30

6

**head xs** es el primer elemeto de la lista xs

head [3,2,5]

3

head “hello”

‘h’

**init xs** es la lista obtenida eliminando el ultimo elemento

init [3,2,7]

[3,2]

# Haskell – Funciones Básicas predefinidas

**iterate f x** es la lista [x. f(x), f(f(x)),...]

take 10 (interate (*2) 1)

[1,2,4,8,16,32,64,128,256,512]

**last xs** es el último elemento de la lista xs

last [3,2,5]

5

**lcm x y** es el mínimo común múltiplo de x e y

lcm 12 30

60

# Haskell – Funciones Básicas predefinidas

**length xs** es el número de elementos de la lista xs

length[4,2,5]

3

**map f xs** es la lista obtenida aplicando f a cada elemento de xs

map (^2) [3,10,5]

[9,100,25]

map even [3,10,5]

[false,true,false]

# Haskell – Funciones Básicas predefinidas

**max x y** es el máximo de x e y

max 3 7

7

**maximum xs** es el máximo elemento de la lista xs

maximum [3,5,2,1]

5

**min x y** es el mínimo de x e y

min 3 7

3

**minimum xs** es el mínimo elemento de la lista xs

minimum [2,3,5,1]

1

# Haskell – Funciones Básicas predefinidas

**notElem x ys** verifica si x no pertenece a ys

notElem 4 [5,3,7]

True

**null xs** verifica si xs es una lista vacia

null []

True

**odd x** verifica so x es impar

odd 23

true

# Haskell – Funciones Básicas predefinidas

**product xs** es el producto de la lista xs

product [2,5,3]

30

**repeat x** repite el valor x

take 5 (repeat 2)

[2,2,2,2,2]

**replicate n x** es la lista formado por n veces el elemento x

replicate 5 2

[2,2,2,2,2]

**reverse xs** es la inversa de la lista xs

Reverse [3,5,2,4]

# Haskell – Funciones Básicas predefinidas

**round x** es el redondeo de x al entero más cercano

round 4.7

5

**show x** es la representación de x como cadena

show 325

“325”

**signum x** es 1 si x es positivo, 0 si x es 0 y -1 si x es negativo

signum 32

1

# Haskell – Funciones Básicas predefinidas

**snd p** es el segundo elemento del par p

snd (3,5)

5

**sqrt x** es la raíz cuadrada de x

sqrt 16

4.0

**sum xs** es la suma de la lista numérica xs

sum [3,2,5]

10

# Haskell – Funciones Básicas predefinidas

**tail xs** es la lista obtenida eliminando el primer elemento de xs

tail [3,2,5]

[2,5]

**truncate x** es la parte entera de x

truncate 34.2

34

# Haskell – Funciones Básicas predefinidas

- `dropWhile p xs` borra el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.

```
λ> dropWhile even [4,8,6,5,0,2,7]
[5,0,2,7]
λ> dropWhile (<7) [4,3,9,1,0,2,7]
[9,1,0,2,7]
```

- `not x` es la negación lógica del booleano `x`.

```
λ> not (2+3 == 5)
False
λ> not (2+3 /= 5)
True
```

# Haskell – Funciones Básicas predefinidas

- `take n xs` es la lista de los `n` primeros elementos de `xs`.

```
λ> take 2 [5,7,9,6]
[5,7]
λ> take 5 [1..]
[1,2,3,4,5]
λ> take 5 [1,3..]
[1,3,5,7,9]
```

- `takeWhile p xs` es el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.

```
λ> takeWhile even [4,8,6,5,0,2,7]
[4,8,6]
λ> takeWhile (<7) [4,3,9,1,0,2,7]
[4,3]
```

# Haskell – Funciones Básicas predefinidas

- `until p f x` aplica `f` a `x` hasta que se verifique `p`.

```
λ> until (>1000) (*2) 1
1024
```

- `zip xs ys` es la lista de pares formado por los correspondientes elementos de `xs` e `ys`.

```
λ> zip [3,5,2] [4,7]
[(3,4),(5,7)]
λ> zip [3,5] [4,7,2]
[(3,4),(5,7)]
λ> zip [3,5] "abc"
[(3,'a'),(5,'b')]
```

# Haskell – Funciones Básicas predefinidas

- `zipWith f xs ys` se obtiene aplicando `f` a los correspondientes elementos de `xs` e `ys`.

```
λ> zipWith (+) [3,5,2] [4,1]
[7,6]
```

# Haskell – Funciones Básicas predefinas sobre Listas dentro de Data.List

- `xs \\ ys` es la lista de los elementos de `xs` que no pertencen a `ys`.

```
λ> [3,2,5,7] \\ [5,6,3]
[2,7]
```

- `delete x ys` es la lista obtenida borrando la primera ocurrencia de `x` en `ys`.

```
λ> delete 3 [5,3,7,3,4]
[5,7,3,4]
λ> delete 9 [5,3,7,3,4]
[5,3,7,3,4]
```

- `find p xs` es justo el primer elemento de `xs` que cumple la propiedad `p` y `Nothing` si ninguno la cumple.

```
λ> find even [7,2,5,6]
Just 2
λ> find even [7,1,5,3]
Nothing
```

# Haskell – Funciones Básicas predefinas sobre Listas dentro de Data.List

- `group xs` es la lista de lista obtenidas agrupando los elementos consecutivo de `xs` que son iguales.

```
λ> group [4,4,7,7,7,5,4,4,4]
[[4,4],[7,7,7],[5],[4,4,4]]
```

- `inits xs` es la lista de los prefijos de `xs`.

```
λ> inits [3,2,5]
[[],[3],[3,2],[3,2,5]]
```

- `intersect xs ys` es la intersección de `xs` e `ys`; es decir, los elementos de `xs`que pertenecen a `ys`.

```
λ> intersect [3,2,5,4] [4,7,2,6]
[2,4]
λ> intersect [3,2,5,4] [8,7,1,6]
[]
```

# Haskell – Funciones Básicas predefinas sobre Listas dentro de Data.List

- `isInfixOf xs ys` se verifica si `xs` es una sublista de `ys`.

```
λ> isInfixOf [3,2,5] [3,2,5,7,6]
True
λ> isInfixOf [2,5,7] [3,2,5,7,6]
True
λ> isInfixOf [7,6] [3,2,5,7,6]
True
λ> isInfixOf [3,5] [3,2,5,7,6]
False
```

- `isPrefixOf xs ys` se verifica si `xs` es un prefijo de `ys`.

```
λ> isPrefixOf [3,2] [3,2,5,4]
True
λ> isPrefixOf [2,5] [3,2,5,4]
False
```

- `isSuffixOf xs ys` se verifica si `xs` es un sufijo de `ys`.

```
λ> isSuffixOf [3,2] [4,5,3,2]
True
λ> isSuffixOf [5,3] [4,5,3,2]
False
```

# Haskell – Funciones Básicas predefinas sobre Listas dentro de Data.List

- `nub xs` es la lista obtenida eliminando las repeticiones de `xs`.

```
λ> nub [3,2,5,2,2,7,5]
[3,2,5,7]
```

- `partition p xs` es el par formado por los elementos de `xs` que cumplen la propiedad `p` y por los que no la cumplen.

```
λ> partition even [3,2,4,7,9,6,8]
([2,4,6,8],[3,7,9])
λ> partition even [3,1,5]
([],[3,1,5])
λ> partition odd [3,1,5]
([3,1,5],[])
```

- `permutations xs` es la liata de las permutaciones de `xs`.

```
λ> permutations [3,2,5]
[[3,2,5],[2,3,5],[5,2,3],[2,5,3],[5,3,2],[3,5,2]]
λ> permutations "abc"
["abc","bac","cba","bca","cab","acb"]
```

# Haskell – Funciones Básicas predefinas sobre Listas dentro de Data.List

- `sort xs` es la lista obtenida ordenando los elementos de `xs`.

```
λ> sort [3,2,5,1,7]
[1,2,3,5,7]
λ> sort "Soria"
"Saior"
λ> sort ["en","todo","la","medida"]
["en","la","medida","todo"]
```

- `subsequences xs` es la lista de las sublistas de `xs`.

```
λ> subsequences [3,2,5]
[[],[3],[2],[3,2],[5],[3,5],[2,5],[3,2,5]]
λ> subsequences "abc"
["","a","b","ab","c","ac","bc","abc"]
```

# Haskell – Funciones Básicas predefinas sobre Listas dentro de Data.List

- `transpose xss` es la transpuesta de `xss`.

```
λ> transpose [[3,2,5],[4,1,6]]
[[3,4],[2,1],[5,6]]
λ> transpose [[3,4],[2,1],[5,6]]
[[3,2,5],[4,1,6]]
```

- `union xs ys` es la unión de `xs` e `ys`; es decir, los elementos que pertenecen a `xs` o `ys`.

```
λ> union [3,2,5,4] [4,7,2,6]
[3,2,5,4,7,6]
```

# Haskell – Funciones Básicas predefinidas Data.Array

- `array (1,n) [(i, f i) | i <- [1..n]` es el vector de dimensión `n` cuyo elemento `i`-ésimo es `f i`.

```
λ> array (1,5) [(i,2*i) | i <- [1..5]]
array (1,5) [(1,2),(2,4),(3,6),(4,8),(5,10)]
```

# Haskell - Funciones

En los ejemplos anteriores, se utilizaron funciones estándar, incluidas junto a una larga colección de funciones en un fichero denominado "estándar prelude" que es cargado al arrancar el sistema. Con dichas funciones se pueden realizar una gran cantidad de operaciones útiles. Por otra parte, el usuario puede definir sus propias funciones y almacenarlas en un fichero de forma que el sistema pueda utilizarlas en el proceso de evaluación. Por ejemplo, el usuario podría crear un fichero fichero.hs con el contenido:

```
cuadrado::Integer -> Integer
cuadrado x = x * x
menor::(Integer, Integer) -> Integer
menor (x,y) = if x <= y then x else y
```

# Haskell - Fuciones

## Nombres de funciones

- Los nombres de funciones tienen que empezar por una letra en minúscula. Por ejemplo,
  - `sumaCuadrado`, `suma_cuadrado`, `suma'`
- Las palabras reservadas de Haskell no pueden usarse en los nombres de funciones. Algunas palabras reservadas son

| | | | | | | |
|---|---|---|---|---|---|---|
| `case` | `class` | `data` | `default` | `deriving` | `do` | `else` |
| `if` | `import` | `in` | `infix` | `infixl` | `infixr` | `instance` |
| `let` | `module` | `newtype` | `of` | `then` | `type` | `where` |

- Se acostumbra escribir los argumentos que son listas usando s como sufijo de su nombre. Por ejemplo,
  - `ns` representa una lista de números,
  - `xs` representa una lista de elementos,
  - `css` representa una lista de listas de caracteres.

# Haskell - Funciones

Para poder utilizar las definiciones anteriores, es necesario cargar las definiciones del fichero en el sistema. La forma más simple consiste en utilizar el comando ":load":

? :l fichero.hs

Reading script file "fichero.hs" . . . ? Si el fichero se cargó con éxito, el usuario ya podría utilizar la definición:

? cuadrado (3+4)

49

? cuadrado (menor (3,4))

9

# Funciones

## Qué es la función const

Primero eche un vistazo al tipo de función Const, como se muestra a continuación.

```
Prelude> :t const
const :: a -> b -> a
```

El procesamiento consiste en aceptar dos parámetros y devolver el primer parámetro. Pase números, cadenas y funciones a la función Const como se muestra a continuación. (lo mismo se puede hacer con la función Flip.)

```
Prelude> const 1 2
1
Prelude> const "Hello" "World"
"Hello"
```

# Haskell - Funciones

- Funciones Si a y b son dos tipos, entonces a->b es el tipo de una función que toma como argumento un elemento de tipo a y devuelve un valor de tipo b. Las funciones en Haskell son objetos de primera clase. Pueden ser argumentos o resultados de otras funciones o ser componentes de estructuras de datos. Esto permite simular mediante funciones de un único argumento, funciones con múltiples argumentos. Considérese, por ejemplo, la función de suma (+). En matemáticas se toma la suma como una función que toma una pareja de enteros y devuelve un entero. Sin embargo, en Haskell, la función suma tiene el tipo:
- (+)::Int->(Int->Int)
- (+) es una función de un argumento de tipo Int que devuelve una función de tipo Int->Int. De hecho "(+) 5" denota una función que toma un entero y devuelve dicho entero más 5. Este proceso se denomina currificación (en honor a Haskell B.Curry ) y permite reducir el número de paréntesis necesarios para escribir expresiones. De hecho, no es necesario escribir f(x) para denotar la aplicación del argumento x a la función x, sino simplemente f x.

# Funciones

Se pueden tener listas como parámetros. Aquí también se usa la recursividad

Ejemplo

**mapSucesor**: dada una lista de enteros, devuelve la lista de los sucesores de cada entero.

```
mapSucesor :: [Integer] -> [Integer]
mapSucesor [] = []
mapSucesor (x:xs) = x+1 :  mapSucesor xs
```

# Definición de Funciones – *por composición*

Las funciones sin restricciones se definen de forma simple por *composición*

```
Prelude> sumaDoble x y = 2 * (x + y)
Prelude> sumaDoble 6 8
28
```

Haskell infiere tipo de dato, generalmente sin inconveniente, pero conviene indicarlos con `::` y `->`

```
Prelude> :{
Prelude| sumaDoble :: Integer -> Integer -> Integer
Prelude| sumaDoble x y = 2 * (x + y)
Prelude| :}
```

Las funciones devuelven solo un resultado, siendo el último tipo de dato el tipo del resultado y los anteriores los tipos de los argumentos

# Definición de Funciones – *por composición*

Haskell permite polimofismo de tipos, se usan patrones para definir la misma función para datos de la misma clase

```
Prelude> :{
Prelude| sumaDoble :: (Num a) => a -> a -> a
Prelude| sumaDoble x y = 2 * (x + y)
Prelude| :}
Prelude> sumaDoble 6 8
28
Prelude> sumaDoble 6.5 7.5
28.0
```

Para facilitar la escritura, vamos a usar un archivo y lo leeremos en el intérprete con el comando `:load` o `:l`

```
Prelude> :l factorial.lhs
[1 of 1] Compiling Main   ( factorial.lhs, interpreted )
Ok, one module loaded.
```

# Definición de Funciones – *con condicionales*

Si se necesita evaluar datos para la aplicación, la definición por composición puede hacerse *con condicionales*

```
fact' :: Int -> Int
fact' n = if n > 0 then n * fact' (n - 1)
    else if n == 0 then 1 else error "Negativo"
```

La *indentación* indica que continua el renglón anterior

# Definición de Funciones – *comparación patrones*

Una alternativa es mediante *comparación de patrones*

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Notar que esta versión no controla el ingreso de valores negativos

# Definición de Funciones – *por partes*

Una alternativa a la *comparación de patrones* es la definición *por partes*

```
fact :: Int -> Int
factorial n
    | n == 0 = 1
    | n > 0  = n * factorial (n - 1)
    | otherwise  = error "Negativo"
```

# Uso de la función main

En el siguiente ejemplo de código, que insertaremos en el archivo ***main.hs***, utilizamos la expresión "**let**" para añadir dos variables ("var1" y "var2"). La línea de comandos presenta el resultado en forma de un **mensaje de texto** definido en la expresión "**putStrLn**":

```
main = do
let var1 = 2
let var2 = 3
putStrLn "El resultado de la suma de las dos cifras es:"
print(var1 + var2)
```

# Uso de la función main

```
1  add :: Integer -> Integer -> Integer   --function declaration
2  add x y =  x + y                       --function definition
3  main = do
4  putStrLn "La suma de ambos números es la siguiente:"
5  print(add 3 5)    --calling a function
```

# Evaluación de Funciones

## Ansiosa – *eager evaluation*

- La más utilizada en programación
- Se resuelve lo más inmediatemente posible
- Los argumentos se evalúan y luego se pasan

## Perezosa – *lazy evaluation*

- Se demora la resolución lo más posible
- Solo se evalúa lo necesario
- Permite trabajar con representaciones infinitas
- Puede consumir más memoria
- No realiza más pasos que la evaluación ansiosa

# Uso de Funciones en otras funciones

Este programa divide en dos grupos a los elementos pares e impares de una lista

**Código fuente**

```
par::[Int]->[Int]
par []=[]
par (a:as)|mod a 2==0=a:par as
    |True=par as

impar::[Int]->[Int]
impar []=[]
impar (a:as)|mod a 2/=0=a:impar as
    |True=impar as

pi::[Int]->String
pi xs="Los pares son:"++show(par xs)++" Los impares son:"++show(impar xs)
```

# Funciones intencionales

Si quisiéramos escribir esto en Haskell, podríamos usar algo como `take 10 [2,4..]`. Pero, ¿y si no quisiéramos los dobles de los diez primeros número naturales, sino algo más complejo? Para ello podemos utilizar listas intensionales. Las listas intensionales son muy similares a los conjuntos definidos de forma intensiva. En este caso, la lista intensional que deberíamos usar sería `[x*2 | x <- [1..10]]`. `x` es extraído de `[1..10]` y para cada elemento de `[1..10]` (que hemos ligado a `x`) calculamos su doble. Su resultado es:

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

Como podemos ver, obtenemos el resultado deseado. Ahora vamos a añadir una condición (o un predicado) a esta lista intensional. Los predicados van después de la parte donde enlazamos las variables, separado por una coma. Digamos que solo queremos los elementos que su doble sea mayor o igual a doce:

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

# Funciones Intencionales

Podemos incluir varios predicados. Si quisiéramos todos los elementos del 10 al 20 que no fueran 13, 15 ni 19, haríamos:

```
ghci> [x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

No solo podemos tener varios predicados en una lista intensional (un elemento debe satisfacer todos los predicados para ser incluido en la lista), sino que también podemos extraer los elementos de varias listas. Cuando extraemos elementos de varias listas, se producen todas las combinaciones posibles de dichas listas y se unen según la función de salida que suministremos. Una lista intensional que extrae elementos de dos listas cuyas longitudes son de 4, tendrá una longitud de 16 elementos, siempre y cuando no los filtremos. Si tenemos dos listas, `[2,5,10]` y `[8,10,11]` y queremos que el producto de todas las combinaciones posibles entre ambas, podemos usar algo como:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

# Funciones Intencionales

También podemos utilizar ajuste de patrones con las listas intensionales. Fijate:

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

# Haskell - Listas

- Listas Si a es un tipo cualquiera, entonces [a] representa el tipo de listas cuyos elementos son valores de tipo a. Hay varias formas de escribir expresiones de listas:

- La forma más simple es la lista vacía, representada mediante [].

- Las listas no vacías pueden ser construidas enunciando explícitamente sus elementos (por ejemplo, [1,3,10]) o añadiendo un elemento al principio de otra lista utilizando el operador de construcción (:).
- Estas notaciones son equivalentes: [1,3,10] = 1:[3,10] = 1:(3:[10]) = 1:(3:(10:[]))
- El operador (:) es asociativo a la derecha, de forma que 1:3:10:[] equivale a (1:(3:(10:[]))), una lista cuyo primer elemento es 1, el segundo 3 y el último 10.

# Haskell - Listas

- Incluye un amplio conjunto de funciones de manejo de listas, por ejemplo:

length xs devuelve el número de elementos de xs

xs ++ ys devuelve la lista resultante de concatenar xs e ys

concat xss devuelve la lista resultante de concatenar las listas de xss map f xs devuelve la lista de valores obtenidos al aplicar la función f a cada uno de los elementos de la lista xs.

Ejemplos:

? length [1,3,10]

3

? [1,3,10] ++ [2,6,5,7]

[1, 3, 10, 2, 6, 5, 7]

? concat [[1], [2,3], [], [4,5,6]]

[1, 2, 3, 4, 5, 6]

? map fromEnum ['H', 'o', 'l', 'a']

[104, 111, 108, 97]

# Uso de patrones en Listas

Si quisiéramos ligar, digamos, los tres primeros elementos de una lista a variables y el resto a otra variable podemos usar algo como `x:y:z:zs`. Sin embargo esto solo aceptará listas que tengan al menos 3 elementos.

Ahora que ya sabemos usar patrones con las listas vamos a implementar nuestra propia función `head`.

```
head' :: [a] -> a
head' [] = error "¡Hey, no puedes utilizar head con una lista vacía!"
head' (x:_) = x
```

Comprobamos que funciona:

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

# Haskell - Tuplas

- Tuplas Si t1, t2, ..., tn son tipos y n>=2, entonces hay un tipo de n-tuplas escrito (t1, t2, ..., tn) cuyos elementos pueden ser escritos también como (x1, x2,..., xn) donde cada x1, x2, ...,xn tiene tipos t1,t2, ..., tn respectivamente.

Ejemplo:

(1, [2], 3) :: (Int, [Int], Int)

('a', False) :: (Char, Bool)

((1,2),(3,4)) :: ((Int, Int), (Int, Int))

- Obsérvese que, a diferencia de las listas, los elementos de una tupla pueden tener tipos diferentes. Sin embargo, el tamaño de una tupla es fijo. En determinadas aplicaciones es útil trabajar con una tupla especial con 0 elementos denominada tipo unidad.
- El tipo unidad se escribe como () y tiene un único elemento que es también ().

# Tuplas

Como vemos, se emparejan los elementos produciendo una nueva lista. El primer elemento va el primero, el segundo el segundo, etc. Ten en cuenta que como las duplas pueden tener diferentes tipos, `zip` puede tomar dos listas que contengan diferentes tipos y combinarlas. ¿Qué pasa si el tamaño de las listas no coincide?

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["soy","una","tortuga"]
[(5,"soy"),(3,"una"),(2,"tortuga")]
```

Simplemente se recorta la lista más larga para que coincida con el tamaño de la más corta. Como Haskell es perezoso, podemos usar `zip` usando listas finitas e infinitas:

```
ghci> zip [1..] ["manzana", "naranja", "cereza", "mango"]
[(1,"manzana"),(2,"naranja"),(3,"cereza"),(4,"mango")]
```

# Ecuaciones con guardas

- Ecuaciones con guardas Cada una de las ecuaciones de una definición de función podría contener guardas que requieren que se cumplan ciertas condiciones sobre los valores de los argumentos.

```
minimo x y  |  x <= y = x
            |  otherwise = y
```

- En Haskell, la variable "otherwise" evalúa a "True". Por lo cual, escribir "otherwise" como una condición significa que la expresión correspondiente será siempre utilizada si no se cumplió ninguna condición previa.

# Ecuaciones con guardas

Otro ejemplo muy simple: vamos a implementar nuestra función `max`. Si recuerdas, puede tomar dos cosas que puedan ser comparadas y devuelve la mayor.

```
max' :: (Ord a) => a -> a -> a
max' a b
     | a > b      = a
     | otherwise = b
```

Las guardas también pueden ser escritas en una sola linea, aunque advierto que es mejor no hacerlo ya que son mucho menos legibles, incluso con funciones cortas. Pero para demostrarlo podemos definir `max'` como:

```
max' :: (Ord a) => a -> a -> a
max' a b | a > b = a | otherwise = b
```

¡Arg! No se lee fácilmente. Sigamos adelante. Vamos a implementar nuestro propio `compare` usando guardas.

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
     | a > b      = GT
     | a == b     = EQ
     | otherwise = LT
```

```
ghci> 3 `myCompare` 2
GT
```

# Haskell - Definiciones locales

- Las definiciones locales pueden también ser introducidas en un punto arbitrario de una expresión utilizando una expresión de la

```
let <decls> in <expr>
```

- Por ejemplo:

? let x = 1 + 4 in x*x + 3*x + 1

41

? let p x = x*x + 3*x + 1 in p (1 + 4)

41

# Haskell - EXPRESIONES LAMBDA

- Expresiones lambda Además de las definiciones de función con nombre, es posible definir y utilizar funciones sin necesidad de darles un nombre explícitamente mediante expresiones lambda de la forma:

```
\ <patrones atómicos> -> <expr>
```

Esta expresión denota una función que toma un número de parámetros (uno por cada patrón) produciendo el resultado especificado por la expresión .

Por ejemplo, la expresión: (\x->x*x) representa la función que toma un único argumento entero 'x' y produce el cuadrado de ese número como resultado. Otro ejemplo sería la expresión (\x y->x+y) que toma dos argumentos enteros y devuelve su suma. Esa expresión es equivalente al operador (+):

```
? (\x y->x+y) 2 3
 5
```

# Haskell - Tipos definidos por el usuario

- Sinónimos de tipo se utilizan para proporcionar abreviaciones para expresiones de tipo aumentando la legibilidad de los programas. Un sinónimo de tipo es introducido con una declaración de la forma:

```
type Nombre a1 ... an = expresion_Tipo
```

Donde Nombre es el nombre de un nuevo constructor de tipo de aridad n>=0

a1,..., an son variables de tipo diferentes que representan los argumentos de Nombre

expresion_Tipo es una expresión de tipo que sólo utiliza como variables de tipo las variables a1,..., an.

Ejemplo:

```
type Nombre = String
type Edad = Integer
type String = [Char]
type Persona = (Nombre, Edad)
```

# Haskell - Tipos definidos por el usuario

- Definiciones de tipos de datos Aparte del amplio rango de tipos predefinidos, en Haskell también se permite definir nuevos tipos de datos mediante la sentencia **data**. La definición de nuevos tipos de datos aumenta la seguridad de los programas ya que el sistema de inferencia de tipos distingue entre los tipos definidos por el usuario y los tipos predefinidos.
- Se utilizan para construir un nuevo tipo de datos formado a partir de otros.

Ejemplo:

data Persona = Pers Nombre Edad

juan::Persona

juan = Pers "Juan Lopez" 23

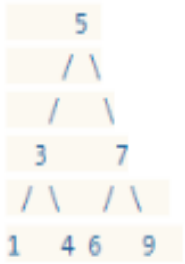
Se debe importar el componente DATA

```
import Data.List (nub)
```

# Arboles

**Definición de tipos recursivos: Los árboles binarios**

- Ejemplo de árbol binario:




- Definición del tipo de árboles binarios:

```
data Arbol = Hoja Int | Nodo Arbol Int Arbol
```

- Representación del ejemplo

```
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))
```

# Arboles

**Definiciones sobre árboles binarios**

- `(ocurre m a)` se verifica si `m` ocurre en el árbol `a`. Por ejemplo,

```
ocurre 4  ejArbol  ==  True
ocurre 10 ejArbol  ==  False
```

```
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d
```

- `(aplana a)` es la lista obtenida aplanando el árbol `a`. Por ejemplo,

```
aplana ejArbol  ==  [1,3,4,5,6,7,9]
```

```
aplana :: Arbol -> [Int]
aplana (Hoja n)     = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
```

# Arboles

**Definiciones sobre árboles binarios**

- Un árbol es ordenado si el valor de cada nodo es mayor que los de su subárbol izquierdo y menor que los de su subárbol derecho.
- El árbol del ejemplo es ordenado.
- `(ocurreEnArbolOrdenado m a)` se verifica si `m` ocurre en el árbol ordenado `a`. Por ejemplo,

```
ocurreEnArbolOrdenado 4  ejArbol  ==  True
ocurreEnArbolOrdenado 10 ejArbol  ==  False
```

```
ocurreEnArbolOrdenado :: Int -> Arbol -> Bool
ocurreEnArbolOrdenado m (Hoja n)  =  m == n
ocurreEnArbolOrdenado m (Nodo i n d)
    | m == n      =  True
    | m < n       =  ocurreEnArbolOrdenado m i
    | otherwise   =  ocurreEnArbolOrdenado m d
```

# Haskell - Tipos Recursivos

- Los tipos de datos pueden autorreferenciarse consiguiendo valores recursivos, por ejemplo:

```
data Expr = Lit Integer
          | Suma Expr Expr
          | Resta Expr Expr
eval (Lit n) = n
eval (Suma e1 e2) = eval e1 + eval e2
eval (Resta e1 e2) = eval e1- eval e2
```

# Recursividad

Ahora implementaremos `sum`. Sabemos que la suma de una lista vacía es 0, lo cual escribimos con un patrón. También sabemos que la suma de una lista es la cabeza más la suma del resto de la cola, y si lo escribimos obtenemos:

```haskell
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

# Recursividad

## Ejemplo de recursión sobre listas

- Especificación: `(sum xs)` es la suma de los elementos de `xs`.
- Ejemplo: sum [2,3,7] $\rightsquigarrow$12
- Definición:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

- Evaluación:

```
      sum [2,3,7]
=     2 + sum [3,7]              [def. de sum]
=     2 + (3 + sum [7])          [def. de sum]
=     2 + (3 + (7 + sum []))     [def. de sum]
=     2 + (3 + (7 + 0))          [def. de sum]
=     12                         [def. de +]
```

# Recursividad

Quicksort en Haskell

```haskell
qsort []     = []
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

Quicksort en C

```c
void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

# Recursión Múltiple

## Recursión múltiple: La función de Fibonacci

- La sucesión de Fibonacci es: 0,1,1,2,3,5,8,13,21,.... Sus dos primeros términos son 0 y 1 y los restantes se obtienen sumando los dos anteriores.
- `(fibonacci n)` es el `n`–ésimo término de la sucesión de Fibonacci. Por ejemplo,

```
fibonacci 8  ⇝  21
```

```
fibonacci :: Int -> Int
fibonacci 0     = 0
fibonacci 1     = 1
fibonacci (n+2) = fibonacci n + fibonacci (n+1)
```

# Uso de acumuladores

## Definición de suma de lista con acumuladores

- Definición de suma con acumuladores:

```
suma :: [Integer] -> Integer
suma = sumaAux 0
  where sumaAux v []     = v
        sumaAux v (x:xs) = sumaAux (v+x) xs
```

- Cálculo con suma:

```
suma [2,3,7] = sumaAux 0 [2,3,7]
             = sumaAux (0+2) [3,7]
             = sumaAux 2 [3,7]
             = sumaAux (2+3) [7]
             = sumaAux 5 [7]
             = sumaAux (5+7) []
             = sumaAux 12 []
             = 12
```

# Uso de Case

```haskell
describeList :: [a] -> String
describeList xs = "La lista es" ++ case xs of []  -> "una lista vacía."
                                              [x] -> "una lista unitaria."
                                              xs  -> "una lista larga."
```

# Haskell - Entrada/Salida

- Una expresión de tipo IO a denota una computación que puede realizar operaciones de Entrada/Salida y devolver un resultado de tipo a. A continuación se declara una sencilla función que muestra por pantalla la cadena "Hola Mundo":

print "Hola, mundo!“

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Read this carefully, because this is your future: " ++ tellFortune name
```

# Entrada / Salida

*Existen varias funciones predefinidas para leer/escribir valores, por ejemplo:*

- ***getChar*** **:** ***IO Char*** *lee un caracter*
- ***putChar*** **:** ***Char*** $\rightarrow$ ***IO*** *() imprime un caracter*
- ***getLine*** **:** ***IO String*** *lee un línea*
- ***putStr*** **:** ***String*** $\rightarrow$ ***IO*** *() imprime una línea*

*A continuación se presenta una función que pide 2 líneas al usuario e indica si son iguales*

```
> compara :: IO ()
> compara = getLine >>= (\s1 ->
>                getLine >>= (\s2 ->
>                if s1 == s2 then putStr "Son iguales"
>                           else putStr "Son diferentes"))
```

*La notación* ***do*** *permite simplificar funciones como la anterior, que podrían re-escribirse como:*

```
> compara2 :: IO ()
> compara2 = do
>              s1 <- getLine
>              s2 <- getLine
>              if s1 == s2 then putStr "Son iguales"
>                          else putStr "Son diferentes"
```

# Importando módulos

```haskell
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

Cuando realizamos `import Data.List`, todas las funciones que `Data.List` exporta están disponibles en el espacio de nombres global. Esto significa que podemos acceder a todas estas funciones desde nuestro script. `nub` es una función que está definida en `Data.List` la cual toma una lista y devuelve otra sin elementos duplicados. Componer `length` y `nub` haciendo `length . nub` produce una función equivalente a `\xs -> length (nub xs)`.

# Creando módulos propios

Como observamos, vamos a calcular el área y el volumen de las esferas, cubos y hexaedros. Continuemos y definamos estas funciones:

```haskell
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 + rectangleArea a c * 2 + rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

# Maneio de archivos

esta es la tercera.

- El procedimiento (`muestraContenidoFichero f`) muestra en pantalla el contenido del fichero `f`. Por ejemplo,

```
λ> muestraContenidoFichero "Ejemplo_1.txt"
Este fichero tiene tres lineas
esta es la segunda y
esta es la tercera.
```

El programa es

```haskell
muestraContenidoFichero :: FilePath -> IO ()
muestraContenidoFichero f = do
  cs <- readFile f
  putStrLn cs
```

# Manejo de archivos

```
escribeArch :: FilePath -> String -> IO ()
escribeArch  f n = do
 writeFile f (n)
```

```
Ok, one module loaded.
*Main>
*Main>
*Main>
*Main>
*Main> escribeArch "archivo1.txt" "esto es una prueba"
*Main> :load
```

# Manejo de archivos

- El procedimiento `(aMayucula f1 f2)` lee el contenido del fichero f1 y escribe su contenido en mayúscula en el fichero f2. Por ejemplo,

```
λ> muestraContenidoFichero "Ejemplo_1.txt"
Este fichero tiene tres lineas
esta es la segunda y
esta es la tercera.

λ> aMayuscula "Ejemplo_1.txt" "Ejemplo_3.txt"
λ> muestraContenidoFichero "Ejemplo_3.txt"
ESTE FICHERO TIENE TRES LINEAS
ESTA ES LA SEGUNDA Y
ESTA ES LA TERCERA.
```

El programa es

```haskell
import Data.Char (toUpper)

aMayuscula f1 f2 = do
  contenido <- readFile f1
  writeFile f2 (map toUpper contenido)
```

# Manejo de archivos

- El procedimiento (`ordenaFichero f1 f2`) lee el contenido del fichero f1 y escribe su contenido ordenado en el fichero f2. Por ejemplo,

```
λ> muestraContenidoFichero "Ejemplo_4a.txt"
Juan Ramos
Ana Ruiz
Luis Garcia
Blanca Perez

λ> ordenaFichero "Ejemplo_4a.txt" "Ejemplo_4b.txt"
λ> muestraContenidoFichero "Ejemplo_4b.txt"
Ana Ruiz
Blanca Perez
Juan Ramos
Luis Garcia
```

El programa es

```haskell
import Data.List (sort)

ordenaFichero :: FilePath -> FilePath -> IO ()
ordenaFichero f1 f2 = do
  cs <- readFile f1
  writeFile f2 ((unlines . sort . lines) cs)
```

# Manejo de archivos

- Las funciones lines y unlines

```
λ> :type lines
lines :: String -> [String]
λ> :type unlines
unlines :: [String] -> String

λ> unlines ["ayer fue martes", "hoy es miercoles","de enero"]
"ayer fue martes\nhoy es miercoles\nde enero\n"
λ> lines it
["ayer fue martes","hoy es miercoles","de enero"]
```

- Las funciones words y unwords

```
λ> :type words
words :: String -> [String]
λ> :type unwords
unwords :: [String] -> String

λ> words "ayer fue   martes"
["ayer","fue","martes"]
λ> unwords it
"ayer fue martes"
```

# Clases en Haskell

## El sistema de clases de Haskell

### Introducción

- Función monomórfica: sólo se puede usar para valores de un tipo concreto

$$
\begin{aligned}
&not &&:: Bool \to Bool\\
&not\ True &&= False\\
&not\ False &&= True
\end{aligned}
$$

- Función polimórfica: se puede usar sobre una familia completa de tipos (aquella que representa el tipo polimórfico)

$$
\begin{aligned}
&id &&:: a \to a\\
&id\ x &&= x\\
&length &&:: [a] \to Int\\
&length[\,] &&= 0\\
&length(x : xs) &&= 1 + length\ xs
\end{aligned}
$$

$$
\begin{aligned}
&id\ 1\\
\Longrightarrow\ &\text{! definición de } id\\
&1
\end{aligned}
\qquad
\begin{aligned}
&id\ True\\
\Longrightarrow\ &\text{! definición de } id\\
&True
\end{aligned}
$$

- ✓ Funciones válidas para cualquier tipo $a$.
- ✓ Siempre se computan igual (definición única).

# Clases en Haskell

- Función sobrecargada:
  - ✓ pueden usarse para más de un tipo, pero con restricciones (**contextos**)
  - ✓ las definiciones (código) pueden ser distintas para cada tipo

EJEMPLO: El operador $(+)$ de Haskell

- puede usarse para sumar enteros
  $1 + 2 \Longrightarrow 3$
- puede usarse para sumar reales
  $1.5 + 2.5 \Longrightarrow 4.0$
- NO puede usarse para sumar booleanos
  $True + False$ "no tiene sentido, no está definido"

► El tipo NO puede ser polimórfico $(+) :: a \longrightarrow a \longrightarrow a$ , ya que

NO se suman del mismo modo valores enteros, reales o racionales

► Deben existir definiciones distintas

- ✓ En HASKELL el tipo de la suma es $(+) :: \underbrace{Num\ a}_{contexto} \Rightarrow a \longrightarrow a \longrightarrow a$
- ✓ Solo se pueden sumar tipos numéricos

# Clases en Haskell

## Clases e Instancias

HASKELL organiza los distintos tipos en clases de tipos.

- **Clase**: conjunto de tipos para los que tiene sentido una serie de operaciones sobrecargadas.

✓ Algunas de las clases predefinidas:

- *Eq* tipos que definen igualdad: $(==)$ y $(\neq)$
- *Ord* tipos que definen un orden: $(\leq)$, $(<)$, $(\geq)$, ...
- *Num* tipos numéricos: $(+)$, $(-)$, $(*)$, ...

- **Instancias**: conjunto de tipos pertenecientes a una clase.

✓ Algunas instancias predefinidas:

- *Eq* tipos que definen igualdad: *Bool*, *Char*, *Int*, *Integer*, *Float*, *Double*, ...
- *Ord* tipos que definen un orden: *Bool*, *Char*, *Int*, *Integer*, *Float*, *Double*, ...
- *Num* tipos numéricos: *Int*, *Integer*, *Float* y *Double*

# Clases en Haskell

## Declaraciones de clases e instancias

- Definición de la clase predefinida $Eq$

```
class Eq a where
    (==) :: a → a → Bool
    (≠)  :: a → a → Bool
```

  - ✓ Las operaciones de una clase se llaman **métodos**
  - ✓ La definición de clase especifica la signatura (tipos) de los métodos
  - ✓ El nombre de la clase comienza por mayúscula

- Para especificar que un tipo implementa los métodos de una clase se usa una definición de instancia:

```
instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False

    x     ≠  y     = not(x == y)
```

```
data Racional = Integer :/ Integer deriving Show

instance Eq Racional where
    (x :/ y) == (x' :/ y') = (x * y' == y * x')

    r        ≠  r'         = not(r == r')
```

# Clases en Haskell

## Métodos por defecto

- Se definen en la clase:

```
class Eq a where
    (==) :: a → a → Bool
    (≠) :: a → a → Bool

    x == y = not (x ≠ y)
    x ≠ y = not (x == y)
```

✓ Para realizar una instancia de $Eq$ basta con definir $(==)$ o $(\neq)$.

✓ Al menos hay que definir una (definiciones mutuamente recursivas).

```
instance Eq Bool where
    True ==True  = True
    False ==False = True
    _     ==_     = False
```

# Subclases en Haskell

## Subclases

- La clase predefinida $Ord$ es **subclase** de la clase $Eq$:

```
class Eq a => Ord a where
  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
  ...

data Ordering = LT | EQ | GT deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

✓ Toda instancia de la subclase debe ser instancia de la clase padre: es un error de compilación incluir un tipo en $Ord$ sin incluirlo en $Eq$.

✓ Lo contrario es posible: hacer un tipo instancia tan solo de $Eq$.

✓ Una subclase tiene visibilidad sobre los métodos de la clase padre: se pueden usar los métodos de $Eq$ en los métodos por defecto de $Ord$

# Clases en Haskell

## Algunas clases predefinidas

- Tipos con igualdad

```
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
  -- Mínimo a implementar: (==) o bien (≠)
  x==y = not(x ≠ y)
  x ≠ y = not(x==y)
```

- Tipos con orden

```
class Eq a ⇒ Ord a where
  compare :: a → a → Ordering
  (<), (≤), (≥), (>) :: a → a → Bool
  max, min :: a → a → a
  -- Mínimo a implementar: (≤) o compare
  compare x y|x==y = EQ
             |x≤y  = LT
             |otherwise = GT
  x≤ y = compare x y ≠ GT
  x <y = compare x y ==LT
  x≥ y = compare x y ≠ LT
  x >y = compare x y ==GT
  max x y |x≥y = x
          |otherwise = y
  min x y |x≤y = x
          |otherwise = y
data Ordering = LT|EQ|GT
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

# Clases en Haskell

## Derivación de instancias

- La claúsula **deriving** permite generar instancias de ciertas clases predefinidas de forma automática.
- ✓ Aparece al final de la declaración de tipo

```
data Color = Rojo | Amarillo | Azul | Verde deriving (Eq, Ord, Show)
```

```
MAIN> Rojo == Verde
False :: Bool
MAIN> Rojo < Verde
True :: Bool
```

```
MAIN> show Rojo
"Rojo" :: String
```

- ► Las instancias generadas usan igualdad estructural: Dos valores son iguales si tienen la misma forma

```
data Racional = Integer :/ Integer deriving Eq
```

genera

```
instance Eq Racional where
  x :/ y == x' :/ y' = (x == x') && (y == y')
```

La igualdad estructural no es adecuada en este caso

```
MAIN> 1 :/ 2 == 2 :/ 4
False :: Bool
```

```
data Nat = Cero | Suc Nat deriving Eq
```

genera

```
instance Eq Nat where
  Cero  == Cero  = True
  Suc x == Suc y = (x == y)
  _     == _     = False
```

La igualdad estructural es adecuada en este caso

# Clases en Haskell

Las relaciones entre contextos dan lugar a una jerarquía de clases

```
class  A a where
   f :: a -> Bool
class  B a where
    g :: A a => a -> a -> a
    g x y  | f x = y
           | otherwise = x
```

- Siendo el tipo de la función g :
- g :: ( A  a , B  a) =>  a  -> a -> a

# Clases en Haskell

Podemos restringir el tipo a de la clase B en el contexto de A a

```
class A a => B a where
  g :: a -> a -> a
  g x y | f x  = y
        | otherwise = x
```

- Siendo ahora el tipo de g :
- `g :: B a => a -> a -> a`
- En este caso se dice que B es una subclase de A.