Programación Declarativa

Francisco Javier López Fraguas

Picoteo de aquí y allá: R. Pinero (UPM), G. Hutton (U. Nottingham), M. Hanus (U. Kiel), ...

DSIC, FdI, UCM, Curso 2016-17

Programación imperativa vs. declarativa

Hace muchos años. . .

1930's: Nace la Teoría de la Computabilidad

Nociones teóricas, no había ordenadores!

- Máquinas de Turing

dispositivo de cómputo $\rightsquigarrow$ arquitectura Von-Neumann
programación imperativa

Programación imperativa vs. declarativa

Hace muchos años. . .

1930's: Nace la Teoría de la Computabilidad

Nociones teóricas, no había ordenadores!

- Máquinas de Turing

  dispositivo de cómputo $\rightsquigarrow$ arquitectura Von-Neumann
  programación imperativa

- λ-cálculo (Church), funciones recursivas (Gödel, Kleene)

  formalismos abstractos $\rightsquigarrow$ definición de funciones
  programación funcional

Hace aún más años ...

- s. IV a.C - s. XIX: Lógica (más o menos formal)
  - Soporte de los aspectos deductivos del conocimiento humano
- 1870-1920: Lógica primer orden
  - Soporte (casi) universal del razonamiento matemático
- 1950: Lógica de Horn
  - Fragmento de la lógica de primer orden fácilmente mecanizable
- 1970-80's: $\rightsquigarrow$ **Programación lógica**
  - Se empieza a explotar el valor computacional de las teorías lógicas y los procesos deductivos
  - Surge Prolog, un lenguaje de programación basado directamente en la lógica de Horn

Alan J. Robinson (uno de los padres de la programación lógica)

La visión de la computación a lo Turing/VonNeumann pone énfasis en la actividad de los cómputos ('cómo'), mientras que el punto de vista declarativo pone énfasis en el resultado de los cómputos ('qué')

Qué hace este código?

```
procedure hazalgo(l,r:index);
var i,j:index; x,w:item
begin
  i := l; j := r;
  x := a[(l+r) div 2];
  repeat
   while a[i] < x do i := i+1;
   while x < a[j] do j := j-1;
   if i <= j then
   begin
    w := a[i]; a[i] := a[j]; a[j] := w;
    i := i+1; j := j-1
   end
  until i > j;
  if l < j then hazalgo(l,j);
  if i < r then hazalgo(i,r);
end
```

Qué hace este código?

```
procedure quicksort(l,r:index);
var i,j:index; x,w:item
begin
  i := l; j := r;
  x := a[(l+r) div 2];
  repeat
   while a[i] < x do i := i+1;
   while x < a[j] do j := j-1;
   if i <= j then
   begin
    w := a[i]; a[i] := a[j]; a[j] := w;
    i := i+1; j := j-1
   end
  until i > j;
  if l < j then quicksort(l,j);
  if i < r then quicksort(i,r);
end
```

Qué hace este código?

```
procedure quicksort(l,r:index);
var i,j:index; x,w:item
begin
  i := l; j := r;
  x := a[(l+r) div 2];
  repeat
   while a[i] < x do i := i+1;
   while x < a[j] do j := j-1;
   if i <= j then
   begin
     w := a[i]; a[i] := a[j]; a[j] := w;
     i := i+1; j := j-1
   end
  until i > j;
  if l < j then quicksort(l,j);
  if i < r then quicksort(i,r);
end
```

Versión declarativa

```
qsort   []       =   []
qsort   (x:l)    =
  qsort [y | y <- l, y < x]
          ++ [x] ++
  qsort [y | y <- l, y > x]
```

Qué hace este código?

```
procedure quicksort(l,r:index);
var i,j:index; x,w:item
begin
  i := l; j := r;
  x := a[(l+r) div 2];
  repeat
   while a[i] < x do i := i+1;
   while x < a[j] do j := j-1;
   if i <= j then
   begin
    w := a[i]; a[i] := a[j]; a[j] := w;
    i := i+1; j := j-1
   end
  until i > j;
  if l < j then quicksort(l,j);
  if i < r then quicksort(i,r);
end
```

Versión declarativa

```
qsort   []     =   []
qsort   (x:l)  =
  qsort [y | y <- l, y < x]
       ++ [x] ++
  qsort [y | y <- l, y > x]
```

- No hay noción de variable asignable
- No hay noción de cambio de estado
- Recursión en lugar de iteración

Sir Anthony Hoare – Premio Turing 1980
(primer Doctor Honoris Causa de UCM en Informática, mayo 2013)

... There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

Richard OKeefe – guru de Prolog, autor de *The Craft of Prolog*

Elegance is not optional. *There is no tension between writing a beautiful program and writing an efficient program. If your code is ugly, the chances are that you either don't understand your problem or you don't understand your programming language, and in neither case does your code stand much chance of being efficient. In order to ensure that your program is efficient, you need to know what it is doing, and if your code is ugly, you will find it hard to analyse.*

Dos paradigmas de programación declarativa (I)

Programación funcional

- Programas $\equiv$ definiciones de funciones
- Cómputos $\equiv$ evaluación de expresiones
- Lenguajes representativos: **Haskell**, Lisp, Scheme, ML, Caml, OCaml, Clean, Erlang, Scala, ...

Características que los distinguen

- Método de evaluación: impaciente, perezosa
- Tipos: Tipado estático, tipado dinámico
- Énfasis en la concurrencia
- Características de OO

Dos paradigmas de programación declarativa (II)

Programación lógica

- Programas $\equiv$ definiciones axiomáticas de relaciones
- Cómputos $\equiv$ deducciones para resolver objetivos
- Lenguajes representativos: **Prolog**, Oz, Mercury, λ-Prolog, Visual Prolog, Curry, ...

Características que los distinguen de Prolog

- Combinación con otros paradigmas
- Tipos
- Orden superior

☞ **Dos partes del curso**

Primera parte

Programación funcional

Lenguaje Haskell

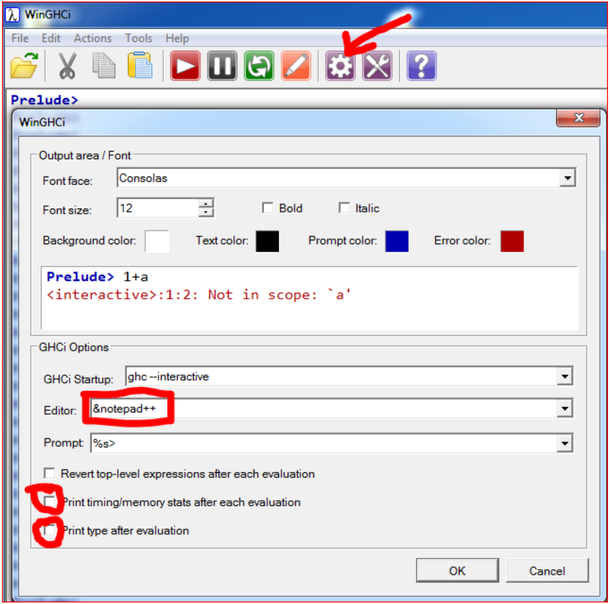
Lenguaje Haskell
(Haskell B. Curry: lógico-matemático 1900-1982)

- www.haskell.org (o googlear 'haskell')
- www.haskell.org/haskellwiki/Introduction
- Descarga del sistema: www.haskell.org/platform/
- Haskell wiki book: en.wikibooks.org/wiki/Haskell
- A Gentle Introduction to Haskell (version 98)
  www.haskell.org/tutorial/index.html
- Haskell report 2010 (definición oficial de Haskell)
  http://www.haskell.org/haskellwiki/Language_and_library_specification
- G. Hutton: *Programming in Haskell*
- B. Ruiz y otros: *Razonando con Haskell*
- R: Bird: *Introducción a la Prog. Funcional con Haskell*

Haskell: primer contacto

Cómputos $\equiv$ evaluación de expresiones

Configura el intérprete a tu gusto



Cómputos ≡ evaluación de expresiones (II)

```
> 29^25
3630362123627258663193028251474330749
> div 8 3
2
> 8/3
2.6666666666666665
> True && (False || not False)
True
> not (not True)
True
```

Enteros, de varios tipos

Reales, de varias precisiones

Booleanos

Constantes en mayúscula

Funciones y variables en minúscula

Operadores infijos (`^`, `/`, `&&`, `||`)

Argumentos de las funciones sin paréntesis

☞notación *currificada* (ver `div` y `not`)

Paréntesis para anidamiento de funciones

(y para inhibir prioridades de operadores)

Cómputos ≡ evaluación de expresiones (III)

```
> true && False
Not in scope:
'true'
```

`true` variable sin definir

Error en tiempo de compilación

```
> div 1 0
Exception
```

Expresión con valor no definido

Error en tiempo de ejecución

```
> 1 + True
Error de tipo
```

Las expresiones deben estar bien tipadas

Error en tiempo de compilación

```
> False && div 1 0 == 2
False
```

En Haskell las expresiones solo se evalúan si hace falta

☞ Evaluación perezosa (*lazy evaluation*)

```
> False && 0
Error de tipo
```

El tipado es estático

```
> False && div 1 0 == 'a'
Error de tipo
```

El tipado es estático

Cómputos $\equiv$ evaluación de expresiones (IV)

```
> 3+3 == 2*3
True
> True && False == True
False
> 3+3 == True
Error de
tipo
```

`==` función de igualdad, polimórfica

Los dos lados de `==` han de tener el mismo tipo

```
> 3+3 /= 4
True
> 3+3 /= True
Error de
tipo
```

`/=` función de desigualdad, polimórfica

Los dos lados de `/=` han de tener el mismo tipo

Funciones predefinidas sobre enteros, reales, booleanos,...

Muchas de ellas están sobrecargadas para distintos tipos
$\rightsquigarrow$ Más explicaciones al ver *clases de tipos*

- `+, -, *, /, div, mod, ^, ^^, **,...`
- `even, odd, lcm, gcd`
- `abs, signum, negate, min, max`
- `pi, exp, sqrt, log, **, logBase, sin, tan, cos, asin, atan, acos, ...`
- `truncate, round, ceiling, floor, fromInteger, toInteger, fromIntegral`

Probadlas todas!

Evaluación de expresiones: tuplas

Tuplas ≡ agrupación de un número fijo de valores de cualquier tipo
$\rightsquigarrow$ tipo de datos polimórfico

```
> (1+2,True && False)
(3,False)    Las componentes pueden ser de distinto tipo
> (1+2,(0,1),0,succ 'a')
(3,(0,1),0,'b')     Hay tuplas de cualquier tamaño
> ()                    Incluso tupla vacía
()
> fst (3,5)      fst tiene un argumento, no dos
3
> snd(5,True)     snd tiene un argumento, no dos
True
> (3,5) == (5,3)
False      El orden influye en el valor de una tupla
> (3,True) == (True,3)
Error de tipo       El orden influye en el tipo de una tupla
> (3,5) == (3,5,5)
Error de
tipo       Tuplas de distinto tamaño tienen tipos distintos
```

Evaluación de expresiones: listas

Listas

- Secuencias de longitud no prefijada   !incluso infinita!
- Los elementos de la lista pueden ser de cualquier tipo
  Listas $\rightsquigarrow$ tipo de datos polimórfico
- Pero todos los elementos deben ser del mismo tipo

```
> [1+2,3,5-1,7]
[3,3,4,7]
> [True,False || True]
[True,True]
> []
[]
> [[1,2],[],[5]]
[[1,2],[],[5]]
> [True, 1, True]
Error de tipo
```

Lista de enteros

Lista de booleanos

Lista vacía (de cualquier tipo)

Lista de listas de enteros

Listas polimórficas pero homogéneas

Evaluación de expresiones: listas (II)

```
> [1,2,3] == [3,4]
False     Listas distintas, pero del mismo tipo
> [1,2] == ['a','b']
Error de tipo   Listas de distinto tipo
> [1,1] == [1]
False     Las repeticiones cuentan
> [0,1] == [1,0]
False     El orden cuenta
```

Evaluación de expresiones: listas (III)

```
> head [1,3,5]
1
```
cabeza de la lista

```
> tail [1,2,3,4]
[2,3,4]
```
resto de la lista

```
> tail [1]
[]
```
Lista vacía de enteros

```
> tail [True]
[]
```
Lista vacía de booleanos

`[]` es una constante polimórfica

```
> tail [1] == tail [True]
Error de tipo
```
Las dos apariciones de `[]` tienen tipos distintos

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
> head [3 `div` 0,7]
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
> head [1,5,3 `div` 0,7]
```

Inciso: ``3 `div` 0`` $\equiv$ `div 3 0`

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `` 3 `div` 0 `` $\equiv$ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
```

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` ≡ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
```

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
[5,Exception
```

Construcción monótona del resultado

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
[5,Exception
```

Construcción monótona del resultado

```
> tail [5,3 `div` 0,7]
[Exception
```

Construcción monótona del resultado

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
[5,Exception
```

Construcción monótona del resultado

```
> tail [5,3 `div` 0,7]
[Exception
```

Construcción monótona del resultado

```
> tail [1,tail [],4]
```

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
[5,Exception
```

Construcción monótona del resultado

```
> tail [5,3 `div` 0,7]
[Exception
```

Construcción monótona del resultado

```
> tail [1,tail [],4]
Error de tipo
```

`tail []` tiene el tipo de una lista

Evaluación de expresiones: listas (IV)

```
> head []
Exception
> tail []
Exception
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

```
> head [3 `div` 0,7]
Exception: divide by zero
```

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

```
> head [1,5,3 `div` 0,7]
1
```

Haskell realiza evaluación perezosa

```
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
[5,Exception
```

Construcción monótona del resultado

```
> tail [5,3 `div` 0,7]
[Exception
```

Construcción monótona del resultado

```
> tail [1,tail [],4]
Error de tipo
```

`tail []` tiene el tipo de una lista

```
> head (tail (tail [1,head [],4]))
```

Evaluación de expresiones: listas (IV)

```
 > head []
 Exception
 > tail []
 Exception
> head [3 `div` 0,7]
Exception: divide by zero
> head [1,5,3 `div` 0,7]
1
> tail [3 `div` 0,1,5,7]
[1,5,7]
> tail [1,5,3 `div` 0,7]
[5,Exception
> tail [5,3 `div` 0,7]
[Exception
> tail [1,tail [],4]
Error de tipo
> head (tail (tail [1,head [],4]))
4
```

`head` y `tail` definidas solo para listas no vacías

Aplicarlas a `[]` no es error sintáctico ni de tipo

`head` y `tail` son funciones *parciales*

Inciso: `3 `div` 0` $\equiv$ `div 3 0`

Haskell realiza evaluación perezosa

Construcción monótona del resultado

Construcción monótona del resultado

`tail []` tiene el tipo de una lista

evaluación perezosa

Tipos: cuestiones básicas

`>:t` *expresion* $\leadsto$ muestra el tipo de *expresion*, sin evaluarla

`e::τ` $\leadsto$ indica que *e* tiene tipo τ

```
>:t True
True::Bool
```

```
>:t 'a'
'a'::Char
```

```
>:t ['a','b','c']
['a','b','c']::[Char]
```

```
>:t 1 == 2
1 == 2::Bool
```

```
>:t "h"++"ola"
"h"++"ola"::[Char]
```

```
>:t not
not::Bool -> Bool
```

$\leadsto$ Tipo funcional

```
>:t (&&)
(&&)::Bool -> Bool -> Bool
```

Tipos: cuestiones básicas (II)

Los tipos no intervienen en los cómputos

- `:t` es un comando del intérprete, no una función (no sirve para formar expresiones)
- `Bool`, `Char`, ..., son tipos, no valores (datos). No se puede formar expresiones con ellos.

```
> Bool == Char
Error:  Bool, Char, not in scope
```

Tipos: cuestiones básicas (III)

Tipos básicos

`Char` `Bool` `Int` `Integer` `Float` `Double` ...

Tipo(s) tupla

`(Char,Int)` `(Int,Int,Int)` `()` ...

Tipo(s) lista

`[Bool]` `[(Char,Int)]` `[[Int]]` ... ~~[Int,Int,Int]~~ ~~[]~~

Tipos funcionales

`Bool -> Bool` `Int -> Char` Funciones de aridad 1

`Bool -> Bool -> Bool` Función de aridad 2

☞Todos estos son tipos monomórficos

Tipos: cuestiones básicas (IV)

Tipos polimórficos (polimorfismo *paramétrico*)

- Contienen variables de tipo (parámetros del tipo), que representan tipos cualesquiera.
- Cada valor concreto de los parámetros determina una instancia concreta del tipo polimórfico.
- El usuario podrá definir nuevos tipos (polimórficos o no)

```
>:t head
head::[a] -> a
```

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

```
>:t []
[]::[a]
```

- Expresiones polimórficas
- `head` puede aplicarse a una expresión e si e`::[`τ`]`, siendo τ un tipo cualquiera, y (`head` e) tendrá entonces el tipo τ

Tipos: cuestiones básicas (V)

Tipos cualificados (polimorfismo *ad hoc* $\rightsquigarrow$ **clases de tipos**)

- Contienen variables de tipo restringidas a pertenecer a una (o varias) cierta familia de tipos (*clase de tipos*)

```
>:t (+)
(+)::Num a => a -> a -> a
```

- + puede aplicarse a e_1 y e_2 si $e_1 :: \tau$ y $e_2 :: \tau$, siendo τ un tipo cualquiera que satisfaga la restricción `Num` τ, es decir, que τ sea un tipo de la clase de tipos `Num` (que es una clase de tipos predefinida).
- ¿Qué tipos están en (o son instancia de) la clase `Num`? `Int, Integer, Float, Double, ...`
- El usuario podrá definir sus propias clases de tipos, e instancias de ellas, así como nuevas instancias de clases ya existentes.

Tipos: cuestiones básicas (V)

Otros ejemplos de tipos cualificados

```
>:t (==)
(==)::Eq a => a -> a -> Bool
```

- Eq es una clase de tipos (no un tipo)
- En Eq están casi todos los tipos

```
>:t (<=)
(<=)::Ord a => a -> a -> Bool
```

- Ord es una clase de tipos (no un tipo)
- En Ord están casi todos los tipos

```
>:t succ
succ::Enum a => a -> a
```

En GHCi: `>:info` *nombre* muestra información de *nombre*

Listas: Funciones de `Prelude`

- Cabeza y resto de una lista

  `head::[a] -> a` , `tail::[a] -> [a]`

  No muy usadas, la verdad...

- Último elemento de una lista no vacía

  `last::[a] -> a`

  ```
  > last [1,3,5,7]
  7
  ```

  ```
  > last []
  Exception
  ```

- Todos menos el último elemento de una lista no vacía

  `init::[a] -> [a]`

  ```
  > init [1,3,5,7]
  [1,3,5]
  ```

  ```
  > init []
  Exception
  ```

- Test de vacuidad de una lista

  `null::[a] -> Bool`

  ```
  > null [1,3]
  False
  ```

  ```
  > null []
  True
  ```

Listas: más funciones de `Prelude`

- Longitud de una lista

```
length::[a] -> Int
```

```
> length [1,3,5,7]
4
```

```
> length []
0
```

- Concatenación de dos listas

```
(++)::[a] -> [a] -> [a]
```

```
> [1,3] ++ [2,4,6]
[1,3,2,4,6]
```

```
> [1] ++ [3] ++ [] ++[1]
[1,3,1]
```

- Inversión de una lista

```
reverse::[a] -> [a]
```

```
> reverse [1,3,5,7]
[7,5,3,1]
```

```
> reverse []
[]
```

Listas: más funciones de Prelude

- Concatenación de los elementos de una lista de listas

```
concat::[[a]] -> [a]
```

```
> concat [[1,3,5],[2,4],[],[6]]
[1,3,5,2,4,6]
```

```
> concat [[]]
[]
```

```
> concat []
[]
```

- Test de pertenencia a una lista

```
elem::Eq a => a -> [a] -> Bool
```

```
> elem 3 [1,2,3]
True
```

```
> elem 3 [1,2]
False
```

```
> elem 3 []
False
```

```
> elem 3 [1 `div` 0]
Exception
```

Listas: más funciones de `Prelude`

- Selección del *n*-simo elemento (contando desde 0)

```
(!!)::[a] -> Int -> a
```

```
> [1,3,5,7] !! 2
5
```

```
> [1,3] !! 2
Exception
```

- Selección de los primeros *n* elementos

```
take::Int -> [a] -> [a]
```

```
> take 2 [1,3,5,7]
[1,3]
```

```
> take 0 [1,3,5,7]
[]
```

```
> take 8 [1,3,5,7]
[1,3,5,7]
```

```
> take 2 []
[]
```

Listas: más funciones de `Prelude`

- Eliminación de los primeros *n* elementos

```
drop::Int -> [a] -> [a]
```

```
> drop 2 [1,3,5,7]
[5,7]
```

```
> drop 0 [1,3,5,7]
[1,3,5,7]
```

```
> drop 4 [1,3,5,7]
[]
```

```
> drop 2 []
[]
```

- Separar los primeros *n* elementos de los demás

```
splitAt::Int -> [a] -> ([a],[a])
```

```
> splitAt 2 [1,3,5,7]
([1,3],[5,7])
```

```
> splitAt 0 [1,3,5,7]
([],[1,3,5,7])
```

Listas: más funciones de `Prelude`

- Sumar (multiplicar) los elementos de una lista de números

```
sum , product::Num a => [a] -> a
```

```
> sum [1,3,5]
9
```

```
> product [1,3,5]
15
```

```
> sum []
0
```

```
> product []
1
```

- Hacer conjunción (disyunción) de los elementos de una lista de booleanos

```
and , or::[Bool] -> Bool
```

```
> and [True,False]
False
```

```
> or [True,False]
True
```

```
> and []
True
```

```
> or []
False
```

Listas: más funciones de `Prelude`

- Emparejar dos listas elemento a elemento

```
zip::[a] -> [b] -> [(a,b)]
```

```
> zip [1,2] ['a','b','c']
[(1,'a'),(2,'b')]
```

```
> zip [1,2] []
[]
```

- Desemparejar una lista de parejas

```
unzip::[(a,b)] -> ([a],[b])
```

```
> unzip [(1,'a'),(2,'b')]
([1,2],['a','b'])
```

```
> unzip []
([],[])
```

Notación [1,2,3] es azúcar sintáctico

`> 1:2:3:[]`	Notación alternativa para listas
`[1,2,3]`	: es una constructora de datos
`> 1:2:3:[] == [1,2,3]`	Comprobamos que son lo mismo!
`True`	: es una constructora de aridad 2
	`[]` es una constructora de aridad 0
	Toda lista es o bien `[]`
	o bien de la forma `x:xs`
	`x` es la *cabeza* y `xs` el *resto*
	`xs` ha de ser otra lista
`> 1:2:3:[] == 1:(2:(3:[]))`	Podemos prescindir de los paréntes
`True`	porque : asocia por la derecha
`> 1:2:3:[] == ((1:2):3):[]`	
`Error de tipo`	
`> 1:2:3:4:[] == 1:2:[3,4]`	Se pueden mezclar notaciones
`True`	

Listas: más funciones de `Prelude`

- Secuencias finitas

```
> [1..4]
[1,2,3,4]
```

```
> [1,3..7]
[1,3,5,7]
```

```
> [1,3..8]
[1,3,5,7]
```

```
> [4..0]
[]
```

```
> [4,3..0]
[4,3,2,1,0]
```

- Secuencias infinitas

```
> [1..]
[1,2,3,......]
```

```
> [1,3..
[1,3,5,......
```

- La evaluación perezosa permite procesar partes finitas

```
> take 2 [1..]
[1,2]
```

```
> [1..]==[0..]
False
```

```
> ([2,7,5]++[0..])!!5
2
```

```
> ([0..]++[2,7,5])!!5
5
```

- Pero no siempre, claro

```
> length [1..]
No termina
```

```
> [1..]==[1..]
No termina
```

Notación [1 . . 4] es azúcar sintáctico

- `[1..4]` $\equiv$ `enumFromTo 1 4`
- `[1,3..7]` $\equiv$ `enumFromThenTo 1 3 7`
- `[1..]` $\equiv$ `enumFrom 1`
- `[1,3..]` $\equiv$ `enumFromThen 1 3`

Expresiones condicionales

`if b then e else e'`

b expresión booleana

e , *e'* expresiones del mismo tipo

```
> if 2+2==4 then [1,2] else []
[1,2]
```

`if _ then _ else _` función de aridad 3,

no es una instrucción de control

```
> [1,if False then 2 else 3,3]
[1,3,3]
```

`if _ then _ else _` puede aparecer en cualquier sitio donde pueda aparecer una expresion

```
> if True then 1 else 3 `div` 0
1
```

evaluación perezosa

```
> if True then 1 else [1,2]
```

Mal tipada

Las partes **then** y **else** deben ser del mismo tipo

El tipo de una expresión se preserva durante su evaluación

Definiciones locales (expresiones o ligaduras *let*)

```
> length [1..10^8]
100000000
(1.70 secs, 4000551148 bytes)
> length [1..10^8] + length [1..10^8]
200000000
(3.39 secs, 8000301264 bytes)
> let x=length [1..10^8] in x+x
```

Definición local de la variable x

```
200000000
```

x se computa una sola vez

```
(1.68 secs, 3999167476 bytes)
```

Hay compartición (*sharing*)

Definiciones locales (II)

`let` x=e `in` e'

x variable ligada	*e* ligadura de *x*	*e'* expresión principal

- Una expresión *let x=e in e'* es una expresión más.
- El valor de *let x=e in e'* es el valor que tenga *e'*.
- En la evaluación de *e'* el valor de *x* es el valor de *e*.
- Valor de *let x=e in e'* = valor de *e'[x/e]*
  donde *e'[x/e]* ≡ resultado de sustituir *x* por *e* en *e'*.
  Pero en *let x=e in e'* se comparte el cómputo de *e* a través de la ligadura de *x* mientras que en *e'[x/e]* se repite (o puede repetirse) el cómputo de *e*.
- El valor de *x* se computa según lo requiera *e'* (evaluación perezosa).
- La ligadura de *x* se circunscribe a *e'*.
- *x* es una variable *muda*. Se puede renombrar (consistentemente) sin riesgo.
- Más adelante: otras variantes de definiciones locales.

Definiciones locales (III)

```
> 5 + let x=2*3 in x+x
17
```
let _ in _ puede aparecer en cualquier sitio
donde pueda aparecer una expresion

```
> 5 + let y=2*3 in y+y
17
```
renombramiento de variables

```
> let x=div 1 0 in fst (1,x)
1
```
evaluación perezosa

```
> let x=length [1..]  in fst
(1,x)
1
```
evaluación perezosa

```
> let x=length [1..10^8] in x+x
200000000
(1.68 secs)
> let x=length [1..10^8] in 3
3
(0.00 secs)
```
evaluación perezosa

Definiciones locales (IV)

```
> (let x=5 in x+x) + (let x=1 in 2*x)
12      ámbito de las ligaduras
> (let x=5 in x+x) + x
Exception:  x not in scope       ámbito de las ligaduras
> let x=2 in let y=x+x in y*y     anidamiento de let's
16
> let y= (let x=2 in x+x) in y*y
16
> let x=2 in let y=x+x in y*y*x
32
> let y= (let x=2 in x+x) in y*y*x
Exception:  x not in scope
> let y=x+x in let x=2 in y*y*x
Exception:  x not in scope
> let x=1 in let x=2 in x
2     ámbito de las ligaduras
> let {x=2 ; y=x+x} in y*y*x      bloque de let's
32
> let {y=x+x ; x=2} in y*y*x      bloque de let's
32
```

Definiciones locales (y V, de momento)

```
> (let x=x in x+x)
```

Ligadura recursiva o circular

`No termina` *(pero no es un error sintáctico)*

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

`x` se liga a [1,2,1,2,...

```
[1,2,1]
```

Ligadura recursiva + evaluación perezosa

Valor de una expresión: nociones y notaciones

- Toda expresión sintácticamente correcta tiene un valor.

  Valor $\equiv$ constante (constructora de datos de aridad 0) o aplicación de una constructorade datos de aridad n a n valores
  (Hay también valores funcionales, que no tenemos en cuenta aquí)

- Notación: $[\![\mathtt{e}]\!] \equiv$ valor (o *denotación* o *semántica*) de e

  En lugar de

  ```
  > 2+2
  4
  ```

  escribiremos $[\![\mathtt{2+2}]\!] = 4$

- Consideramos el valor especial $\perp$ (valor *bottom* o *indefinido*) para aquellos casos en los que la evaluación no termina o genera directamente un error en tiempo de ejecución.

  $[\![\mathtt{head\ []}]\!] = [\![\mathtt{length\ [1..]}]\!] = \perp$

- A los valores que contienen $\perp$ los llamamos valores *parciales*

  $[\![\mathtt{[1+1,div\ 1\ 0]}]\!] = [2,\perp]$ $[\![\mathtt{[1]++head\ []}]\!] = 1{:}\perp$

- Hay valores que son *infinitos*.

  $[\![\mathtt{[1..]}]\!] = [1,2,3,\ldots]$

Valor de una expresión (II)

Orden de información entre valores

- Se define entre valores una relacion $v \sqsubseteq v'$, leída como:

  v es una aproximación a v'
  v está menos definido que v'
  v tiene menos información que v'

- $\sqsubseteq$ es la menor relación de orden que cumple:
  - $\perp \sqsubseteq v$, para todo v
  - $v_1 \sqsubseteq v'_1, \ldots, v_n \sqsubseteq v'_n \Rightarrow c \;\; v_1 \ldots v_n \sqsubseteq c \;\; v'_1 \ldots v'_n$, para cualquier constructora de datos c y valores v_i, v'_i

  Es decir:

  $v \sqsubseteq v'$ si v' resulta de reemplazar por cualquier valor las apariciones de $\perp$ que haya en v

- Esta noción se puede extender también a expresiones

Orden de información: ejemplos

$\perp \sqsubseteq 2$ $\quad (\perp, \perp) \sqsubseteq (3, 2)$ $\quad (\perp, 2) \sqsubseteq (3, 2)$ $\quad (3, \perp) \sqsubseteq (3, 2)$

$1 : \perp : [] \sqsubseteq 1 : 0 : []$ (o sea, $[1, \perp] \sqsubseteq [1, 0]$) $\quad 1 : \perp \sqsubseteq [1]$

$\perp : \perp : \perp \sqsubseteq [1, 2]$ $\quad 1 : 2 : \perp \sqsubseteq [1, 2, 3, \ldots]$

$2 \; \overset{\not\sqsubseteq}{\not\sqsupseteq} \; 3$ $\quad (\perp, 2) \; \overset{\not\sqsubseteq}{\not\sqsupseteq} \; (2, \perp)$ $\quad [\,] \; \overset{\not\sqsubseteq}{\not\sqsupseteq} \; [1, 2] \; \overset{\not\sqsubseteq}{\not\sqsupseteq} \; [1, 2, 3]$ $\quad [1, \perp] \; \overset{\not\sqsubseteq}{\not\sqsupseteq} \; [1, 2, 3]$

Orden de información: ejemplos

$\perp \sqsubseteq 2$ $(\perp,\perp) \sqsubseteq (3,2)$ $(\perp,2) \sqsubseteq (3,2)$ $(3,\perp) \sqsubseteq (3,2)$

$1:\perp:[] \sqsubseteq 1:0:[]$ (o sea, $[1,\perp] \sqsubseteq [1,0]$) $1:\perp \sqsubseteq [1]$

$\perp:\perp:\perp \sqsubseteq [1,2]$ $1:2:\perp \sqsubseteq [1,2,3,\ldots]$

$2 \underset{\not\sqsupseteq}{\not\sqsubseteq} 3$ $(\perp,2) \underset{\not\sqsupseteq}{\not\sqsubseteq} (2,\perp)$ $[\,] \underset{\not\sqsupseteq}{\not\sqsubseteq} [1,2] \underset{\not\sqsupseteq}{\not\sqsubseteq} [1,2,3]$ $[1,\perp] \underset{\not\sqsupseteq}{\not\sqsubseteq} [1,2,3]$

En el orden $\sqsubseteq$

- Hay un elemento mínimo $\perp$
- No hay un elemento máximo
- Los valores *totales* (los que no tienen $\perp$) son elementos maximales

Un par de propiedades interesantes de los programas funcionales (puros)

Transparencia referencial

- En un programa dado, $[\![e]\!]$ es el mismo en cualquier aparición de e (ausencia de efectos laterales)
- Si $[\![e]\!] = [\![e']\!]$ entonces e se puede cambiar por e' en cualquier sitio.

Monotonía de la evaluación

- $e \sqsubseteq e' \Rightarrow [\![e]\!] \sqsubseteq [\![e']\!]$
- Como consecuencia: $[\![e]\!]$ maximal $\wedge\ e \sqsubseteq e' \Rightarrow [\![e]\!] = [\![e']\!]$

☞ Más fácil razonar con los programas funcionales

Programas Haskell

Programación funcional

- Programas $\equiv$ definiciones de funciones
- Cómputos $\equiv$ evaluación de expresiones

Un programa Haskell *file.hs* consta de:

- Definiciones de funciones
- Definiciones acerca de tipos:
  - Nuevos tipos de datos (*data*)
  - Nuevas clases de tipos (*class*)
  - Declaraciones de instancia de tipos (*instance*)
  - Alias de tipo (*type*) y tipos isomorfos (*newtype*)
- Nuevos operadores infijos (*infix*)
- Declaraciones relativas a módulos (*module*, *import*,...)

Definición de nuevas funciones

Las funciones son definidas mediante **ecuaciones**

```
doble x = x + x
factorial n = product [1..n]
sandwich xs ys =  let  us = xs++xs
                       vs = ys++ys
                   in  us++vs++us
```

- ⋆ Notación currificada
- ⋆ `xs,ys,us,vs` nombres típicos para listas

Definición de nuevas funciones

Las funciones son definidas mediante **ecuaciones**

```
doble x = x + x
factorial n = product [1..n]
sandwich xs ys =  let  us = xs++xs
                       vs = ys++ys
                  in   us++vs++us
```

- ⋆ Notación currificada
- ⋆ `xs,ys,us,vs` nombres típicos para listas
- ⋆ ¿No hay marcadores de fin de ecuación?

Definición de nuevas funciones

Las funciones son definidas mediante **ecuaciones**

```
doble x = x + x ;
factorial n = product [1..n] ;
sandwich xs ys =  let  us = xs++xs ;
                       vs = ys++ys
                   in  us++vs++us
```

- ⋆ Notación currificada
- ⋆ `xs,ys,us,vs` nombres típicos para listas
- ⋆ ¿No hay marcadores de fin de ecuación?

  Sí, hay un ; implícito entre ecuaciones

  ☞ regla de indentación

Inciso: regla de indentación

```
doble x = x + x
  factorial n = product [1..n]  Error de sintaxis!
sandwich xs ys = let us =  xs++xs
                           vs = ys++ys in us ++ vs ++ us
```

Lo siguiente sí es correcto (pero no recomendado)!

```
{doble x = x + x ;   factorial n = product [1..n]
     ;
    sandwich xs ys = ...}
```

- En una secuencia de definiciones ecuacionales, hay un ; implícito cada vez que una línea comienza en la misma columna que la definición anterior
- Se aplica también a las secuencias de definiciones locales *let* y *where* (y *do*, que veremos más adelante)

Definición de nuevas funciones (II)

Distinciones de casos por expresiones condicionales

```
factorial n = if n==0  then 1
                       else n*factorial (n-1)
```

```
f x y =  if x==0  then True else
         if y==0  then False
                  else f (x-1) (y-1)
```

- ⋆ Estas definiciones hay que escribirlas en un *file.hs*
- ⋆ No usar tabuladores!

Definición de nuevas funciones (III)

Distinciones de casos por **ajuste de patrones**

```
factorial 0 = 1
factorial n = n*factorial (n-1)
```

```
f 0 y = True
f x 0 = False
f x y = f (x-1) (y-1)
```

En el lado izquierdo de cada ecuación se indica a qué valores de los argumentos resulta aplicable la ecuación

- ⋆ Puede haber más de una ecuación por función
- ⋆ Puede haber *solapamiento* de patrones entre ecuaciones
- ⋆ Una ecuación se aplica solo si las anteriores no son aplicables

$\Longrightarrow$ El orden de las ecuaciones importa

Para bien: ecuaciones más simples (condiciones implícitas)

Para mal: cada ecuación suelta no tiene valor declarativo

⋆ La aparición de patrones de ajuste en las distintas ecuaciones guía la evaluación perezosa de la función

Definición de nuevas funciones (IV)

Distinciones de casos por **ecuaciones guardadas**

```
factorial n
  | n==0   = 1   Guarda
  | True   = n*factorial (n-1)
             Las guardas pueden solaparse
             Uso secuencial de las ecuaciones guardadas
```

```
f x y
  | x==0   = True
  | y==0   = False
  | True   = f (x-1) (y-1)
```

f t_1 ...t_n	t_i patrones *lineales* (sin variables repetidas)
| b_1 = e_1	b_i expresiones booleanas
...	e_i expresiones (del mismo tipo)
| b_m = e_m	Ojo: | b_i = e_i no es una expresión

Variaciones sobre el mismo tema

```
factorial n
  | n==0   = 1
  | n>0    = n*factorial (n-1)
```

Las guardas pueden no ser exhaustivas

```
> factorial (-2)
Exception:  Non-exhaustive patterns
```

Lo mismo ocurre con el ajuste de patrones

Variaciones sobre el mismo tema

```
factorial n
  | n==0  = 1
  | n>0   = n*factorial (n-1)
  | n<0   = error "argumento negativo"
```

```
> factorial (-2)
Exception:  argumento negativo
```

error *string*

Genera un error con mensaje asociado *string*

Variaciones sobre el mismo tema

```
factorial n
   | n==0   = 1
   | n>0    = n*factorial (n-1)
   | n<0    = error "el argumento "++show n++" es negativo"
```

```
> factorial (-2)
Exception:  el argumento -2 es negativo
```

show *x*

Devuelve un string (lista de caracteres) que representa a x

x de cualquier tipo de la clase *Show*

Variaciones sobre el mismo tema

```
factorial n
  | n==0        = 1
  | n>0         = n*factorial (n-1)
  | otherwise   = error "argumento negativo"
```

otherwise

Función de aridad 0 definida en `Prelude`

- ```
  otherwise = True
  ```
- Por tanto, $[\![\texttt{otherwise}]\!] = \mathit{True}$
  ```

Más ajuste con patrones constantes: booleanos

```
not True  = False
not False = True
```

```
infixr 3 &&  -- conjunción
True  && y = y
False && _ = False
```

```
-- También podra ser
True  && True = True
  _   &&   _  = False
```

```
infixr 2 || -- disyunción
False || y = y
True  || _ = True
```

```
(||) False  y = y
(||) True   _ = True
```

Cosas que aprendemos

- Patrones para distinguir casos
- Definición de operador infijo
  `infixr`
  `infixl` *prioridad operador*
  `infix`
  `x && y && z` ≡ `x && (y && z)`
  `x && y || z` ≡ `(x && y) || z`
- Variable anónima `_`: cada aparición es una variable nueva
- Uso prefijo de operadores `(||)` , `(&&)`: a veces es necesario o conveniente

Ajuste de patrones no constantes: tuplas

Con ajuste de patrones

```
fst  (x,_) = x
snd  (_,y) = y
swap (x,y) = (y,x)
```

Sin ajuste de patrones

```
fst xy = ???
snd xy = ???
swap xy = (fst xy , snd xy)
```

Suma de complejos

```
infixl 6 +.
(a,b) +.(c,d) = (a+b,c+d)
```

Sin ajuste

```
z +.z' = (fst z+fst z',snd z+snd z')
```

Cosas que aprendemos

- Ajuste de patrones da acceso a componentes
- Sin ajuste de patrones
  - Necesitamos más primitivas
  - Programas más complejos

Aprenderemos a usar $+$ en lugar del nuevo operador $+.$

```haskell
soluciones' (a,b,c) =
  let d = b^2-4*a*c
      e = -b/2*a
      r = sqrt d/2*a
  in  if d>0  then [e+r,e-r] else
      if d==0 then [e]
              else []
```

```haskell
soluciones (a,b,c)
   | d>0  = [e+r,e-r]
   | d==0 = [e]
   | d<0  = []
   where d = b^2-4*a*c
         e = -b/2*a
         r = sqrt d/2*a
```

Cosas que aprendemos

- Patrones, guardas, defs. locales, condicionales pueden coexistir
- Patrón *(a,b,c)*: determina la forma del argumento y da acceso a sus componentes
- *let* no combina bien con ecuaciones guardadas
- *where*: definiciones locales cuyo ámbito se extiende a un grupo de ecuaciones previo con el mismo lado izquierdo
- *where* no forma parte de las sintaxis de las expresiones (como las guardas)
- *where* puede usarse para una sola ecuación

```haskell
soluciones'' (a,b,c) =
  if d>0  then [e+r,e-r] else
  if d==0 then [e]
          else []
  where {d = b^2-4*a*c ; e = -b/2*a ; r = sqrt d/2*a}
```

Ajuste de patrones con listas

```
-- cabeza y resto de una lista
head ::  [a] -> a
head (x:xs)  = x
tail ::  [a] -> [a]
tail (x:xs) = xs

-- test de lista vacía
null ::  [a] -> Bool
null []     = True
null (_:_) = False

-- longitud de una lista
length ::  [a] -> Int
length [] = 0
length (x:xs) = 1+length xs

-- concatenación de listas
infixr 5 ++
(++) ::  [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

Cosas que aprendemos

- `(x:xs)` $\rightsquigarrow$ patrón genérico de lista no vacía
- Patrones distinguen casos y dan acceso a componentes
- Recursión como mecanismo de control
- Típicamente, la recursión implica recorridos izda-dcha de las listas

Recordemos: declaración de tipos es opcional, pero recomendada

Otros ejemplos de programación con listas

```haskell
-- pertenencia a una lista
elem ::  Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (x':xs) = x==x' || elem x xs

-- n-ésimo elemento de una lista
infixl 9 !!
(!!)  ::  [a] -> Int -> a
(x:_)  !!  0 =  x
(_:xs) !!  n =  xs !!  (n-1)

-- suma de los elementos
sum ::  Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

```haskell
-- Tomar n elementos en cabeza
take ::  Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0 = []
  | otherwise = x:take (n-1) xs

-- Emparejar dos listas
zip ::  [a] -> [b] -> [(a, b)]
zip (x:xs) (y:ys) = (x,y):zip xs ys
zip _ _ = []

-- Último elemento de una lista
last ::  [a] -> a
last [x] = x
last (x:x':xs) = last (x':xs)
```

Otro ejemplo: inversa de una lista

```
--  reverse [x1,...,xn] = [xn,...,x1]
reverse ::  [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Evaluación de `reverse [1,2,3,4]`

```
01  r [1,2,3,4]    ≡ (equivalencia sintctica, no es un paso real)
02  r (1:2:3:4:[]) =
03  r (2:3:4:[])++[1] =
04  (r (3:4:[])++[2])++[1] =
05  ((r (4:[])++[3])++[2])++[1] =
06  (((r []++[4])++[3])++[2])++[1] =
07  ((([]++[4])++[3])++[2])++[1] =
08  (([4]++[3])++[2])++[1] =
09  ((4:([]++[3]))++[2])++[1] =
10  (4:(([]++[3])++[2]))++[1] =
11  4:((([]++[3])++[2])++[1]) =
12  4:(([3]++[2])++[1]) ≡
13  4:(((3:[])++[2])++[1]) =
14  4:((3:([]++[2]))++[1]) =
15  4:(3:(([]++[2])++[1])) =
16  4:(3:([2]++[1])) =
17  4:(3:((2:[])++[1])) ≡
18  4:(3:(2:([]++[1]))) =
19  4:(3:(2:[1])) ≡
20  [4,3,2,1]
```

15 pasos de reducción $\rightsquigarrow$ complejidad cuadrática

`reverse` con complejidad lineal

Usamos una función auxiliar con un argumento adicional que juega el papel de *acumulador*, en el que se va construyendo la inversa de `xs` según se va recorriendo `xs`

```
reverse xs          = revAux xs []
revAux ::  [a] -> [a] -> [a]
revAux []      acc = acc
revAux (x:xs) acc = revAux xs (x:acc)
```

O bien, usando `where` para definir `revAux` localmente:

```
reverse xs = revAux xs []
 where      -- ojo a la indentacin
 revAux ::  [a] -> [a] -> [a]
 revAux []      acc = acc
 revAux (x:xs) acc = revAux xs (x:acc)
```

Evaluación de `reverse` lineal

```
r [1,2,3,4]    ≡ (equivalencia sintctica, no es un paso real)
r (1:2:3:4:[]) =
raux (1:2:3:4:[]) [] =
raux (2:3:4:[]) (1:[]) =
raux (3:4:[]) (2:1:[]) =
raux (4:[]) (3:2:1:[]) =
raux [] (4:3:2:1:[]) =
4:3:2:1:[] ≡
[4,3,2,1]
```

6 pasos de reducción $\leadsto$ complejidad lineal

`raux` usa recursión final (*tail recursion*)

Haskell y los tipos

- Lenguaje estáticamente, fuertemente tipado
- Basado en el sistema de Hindley-Milner (creado para ML)
  ☞ Proporciona polimorfismo paramétrico
- Extendido mediante un sistema de clases de tipos
  ☞ Proporciona polimorfismo *ad-hoc* o sobrecarga
- Otras extensiones: laboratorio de ideas muy activo!

⋆ Robin Milner (1934-2010): Premio Turing 1991, realizó contribuciones esenciales en demostradores automáticos (LCF), lenguajes de programación (ML) y sistemas concurrentes (π-cálculo)
⋆ Roger Hindley: contribuciones a la lógica y el λ-cálculo

Sistema de tipos Hindley-Milner

- Tipo $\equiv$ colección de valores
- Una expresión tiene el tipo de su valor

  `e::T` expresa que `e` tiene o admite el tipo `T`

  En Haskell: `> :t e` muestra el tipo de *e*

En Hindley-Milner

- Cada expresión bien tipada puede admitir varios tipos, pero un solo *tipo principal*, que es el más general de todos ellos.
- Los tipos principales pueden ser inferidos, sin necesidad de que el programador declare los tipos.
- Una propiedad esencial (*preservación de tipos*): el tipo de una expresión no cambia durante el proceso de su evaluación
  - O sea: $e :: T \wedge e \rightarrow e' \;\Rightarrow\; e' :: T$
    donde $e \rightarrow e'$ indica un paso de evaluación
  - $e :: T \;\Rightarrow\; [\![\mathrm{e}]\!] :: T$ aceptando que $\perp :: T$

Anatomía de los tipos Hindley-Milner
Tipos monomórficos

```
T ::=      TP
       |   (T1,...,Tn)
       |   [T]
       |   T -> T'
```

`TP`	Tipo primitivo: Char, Bool, Int,...
`(T1,...,Tn)`	Producto de tipos: T1×...×Tn
`[T]`	Listas de elementos de tipo T
`T -> T'`	Tipo funcional

`Añadiremos tipos definidos por el usuario`

Algunos tipos monomórficos

`Char` `(Char,Int)` `[Int]` `[[Int]]` `Int -> Int`

`(Char->Int , Int , Bool)` `(Char->Int)->([Char]->[Int])`

Anatomía de los tipos Hindley-Milner
Tipos polimórficos

Tipos simples

`TS ::=`		`TP`	Tipo primitivo: Char, Bool, Int,...
	|	`(TS1,...,TSn)`	Producto de tipos: TS1$\times \ldots \times$TSn
	|	`[TS]`	Listas de elementos de tipo TS
	|	`TS -> TS'`	Tipo funcional
	|	`a`	Variable de tipo: `a,a',b,...`

Tipos genéricos o esquemas de tipo

`T ::=`		`TS`	Tipo simple
	|	$\forall$ `a.T`	Tipo cuantificado o polimórfico

O sea: `T` $\equiv \forall a_1 \ldots \forall a_n$.`TS` $(n \geq 0)$

`T` es *cerrado* si todas las variables de `TS` están cuantificadas

Algunos tipos simples

`Char` `(a,b)` `[b]` `Int -> Int` `Int -> a`

`(a , Bool , a')` `(a->b)->([a]->[b])`

Algunos esquemas de tipo

`Char` `∀ b.(a,b)` `[Int->Int]` `∀ a.Int -> a`

`(a , Bool , a')` `∀ a.∀ b.(a->b)->([a]->[b])`

Estos son cerrados

Tipos de algunas expresiones (algunas son funciones)

`'a'::Char` `(True,'z')::(Bool,Char)`

`[['b','\n'],[]]::[[Char]]` `not::Bool -> Bool`

`length::∀a.[a]->Int` `fst::∀a.∀b.(a,b)->a`

Polimorfismo paramétrico

`fst::∀a.∀b.(a,b)->a` se infiere de `fst (x,y) = x`

- `x,y` valores cualesquiera
- `a,b` tipos cualesquiera
- `fst` definida de modo uniforme para todos los tipos

Con las opciones por defecto, Haskell no muestra los $\forall$

Funciones *currificadas*

¿Qué pasa con las funciones de más de un argumento?

¿Cuál es el tipo de `(&&)` , `(++)` , `take` , `elem` , ...?

`(&&)::`	`Bool->Bool->Bool`	$\equiv$ `Bool->(Bool->Bool)`
`(++)::`	$\forall a.$`[a]->[a]->[a]`	$\equiv$ $\forall a.$`[a]->([a]->[a])`
`take::`	$\forall a.$`Int->[a]->[a]`	$\equiv$ $\forall a.$`Int->([a]->[a])`
`elem::`	$\forall a.$`a->[a]->Bool`	$\equiv$ $\forall a.$`a->([a]->Bool)`

☞ Los argumentos vienen de uno en uno
Visión currificada de las funciones

Visión currificada de las funciones

Sea f una función de tres argumentos x, y, z que se toman de tres conjuntos (tipos) A, B, C, y que devuelve un resultado r de un conjunto D.

▷ Visión matemática usual: tres argumentos en una terna

$f : A \times B \times C \ \rightarrow \ D$
$f(x, y, z) = r$

f, aplicada a la terna (x, y, z), devuelve r

▷ Visión currificada: los argumentos de uno en uno

$f : A \rightarrow (B \rightarrow (C \rightarrow D))$
$((f \ x) \ y) \ z = r$

f, aplicada a x, devuelve la función que, aplicada a y, devuelve la función que, aplicada a z, devuelve r

Dos azúcares sintácticos de uso **constante**

La flecha de los tipos (->) asocia por la derecha

```
A -> B -> C -> D  ≡  A -> (B -> (C -> D))
                  ≡  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

```
f x y z  ≡  ((f x) y) z
         ≡  (f x y) z
         ≡  (f x) y z
         ≢  f (x y z)
         ≢  f (x y) z
         ≢  f x (y z)
```

Efectos de la currificación

Un lema famoso de la programación funcional

Las funciones son *ciudadanos de primera clase*

- Una función puede ser argumento o resultado de otra
- Las expresiones de tipo funcional son evaluables

  Si $e :: T \to T'$, entonces $[\![e]\!]$ es una función de T en T'

Efectos de la currificación (II)

Aplicaciones parciales

- Si $f :: T_1 \to T_2 \to \ldots \to T_n \to T$ y $m < n$ entonces $f\ e_1\ \ldots\ e_m$ es una aplicación parcial de f.
  Debe tenerse: $e_1 :: T_1, \ldots, e_m :: T_m$
  Se tendrá: $f\ e_1\ \ldots\ e_m :: T_{m+1} \to \ldots \to T_n \to T$
- `take 2::[a] -> [a]` `take:: Int -> [a] -> [a]`
  `elem::a -> [a] -> Bool`
  `elem (True || False)::[Bool] -> Bool`
  `(&&) False::Bool -> Bool`

Veremos un azúcar para aplicaciones parciales de operadores

Efectos de la currificación (III)

Funciones de orden superior (*HO functions*)

- Funciones con algún argumento (o el resultado) de tipo funcional.
- Favorecen la abstracción, concisión y reutilización de código.

Funciones de primer orden: las "normales", que nos son de OS

Funciones de orden superior

Un par de funciones de primer orden...

```
-- incList xs:  sumar uno a todos los elementos de xs
-- incList [x1,...,xn] = [1+x1,...,1+xn]
inc::[Int] -> [Int]
inc x = 1+x
incList []     = []
incList (x:xs) = inc x:incList xs
```

Recomendado: declarar siempre los tipos

```
-- lengthList xss:  longitudes de los elementos de xss
-- lengthList [xs1,...,xsn] = [length xs1,...,length xsn]
lengthList::[[a]] -> [Int]
lengthList []     = []
lengthList (x:xs) = length x:lengthList xs
```

Se repite la misma estructura:
aplicar una función f a todos los elementos de la lista

Abstracción de OS: `map`

```
-- map f xs = resultado de aplicar f a todos los elementos de xs
-- map f [x1,...,xn] = [f x1,...,f xn]
map::(a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x:map f xs
```

```
> map (take 2) [[1,2,3],[2],[7,6,4]]
[[1,2],[2],[7,6]]
> map not [True,False,False]
[False,True,True]
```

```
-- incList [x1,...,xn] = [1+x1,...,1+xn]
-- lengthList [xs1,...,xsn] = [length xs1,...,length xsn]
incList xs    = map inc xs
lengthList xs = map length xs
```

Inciso: cancelación de argumentos

En lugar de

```
incList xs    = map inc xs
lengthList xs = map length xs
```

Podemos definir de modo equivalente

```
incList      = map inc
lengthList   = map length
```

Pointless programming

Tanto recursión como aplicación a argumentos quedan implícitas

No es un azúcar sintáctico, es que ambas definiciones son equivalentes

```
incList [1,2,3] = map inc [1,2,3] = ...
incList [1,2,3] = map inc [1,2,3] = ...
```

Otro inciso: secciones de operadores infijos

Dado un operador infijo $\bigoplus$, hay sendas notaciones especiales para escribir su aplicación parcial a sus argumentos izquierdo o derecho.

(x $\bigoplus$)

Denota la función definida como `(x ⊕) y = x ⊕ y`

```
> (2 ^) 3
8
```

```
> map (2 ^) [1,2,3]
[2,4,8]
```

Es un azúcar para la aplicación parcial `((⊕) x)`

($\bigoplus$ y)

Denota la función definida como `(⊕ y) x = x ⊕ y`

```
> (^ 2) 3
9
```

```
> map (:[]) [1,2,3]
[[1],[2],[3]]
```

```
incList = map (+ 1)
```

filter

```
-- filter p xs = lista de elementos de xs que cumplen p
filter::(a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
  | p x        = x:filter p xs
  | otherwise  =   filter p xs
```

all, any

```
-- all p xs = todos los elementos de xs cumplen p
-- any p xs = algún elemento de xs cumple p
all _ []        =  True
all p (x:xs)    =  p x && all p xs

any _ []        = False
any p (x:xs)    = p x || any p xs
```

```
> filter (> 1) [-1,3,0,4]
[3,4]
> all (> 1) [-1,3,0,4]
False
> any (> 1) [-1,3,0,4]
True
```

takeWhile, dropWhile, span, break

```
-- takeWhile p xs = mayor prefijo de xs cuyos elementos cumplen p
-- dropWhile p xs = resultado de eliminar (takeWhile p xs) de xs
takeWhile, dropWhile::  (a -> Bool) -> [a] -> [a]
span, break ::  (a -> Bool) -> [a] -> ([a], [a])
takeWhile _ []    = []
takeWhile p (x:xs)
     | p x        = x :  takeWhile p xs
     | otherwise = []

dropWhile _ []    = []
dropWhile p (x:xs)
     | p x        = dropWhile p xs
     | otherwise = x:xs

span p  xs = (takeWhile p xs,dropWhile p xs)
break p xs = span (not.p) xs   -- .  es la composiciÓn de funciones
                               -- xs se podría cancelar, pero p no
```

Composición de funciones: .

```
-- f.g = composición de las funciones f y g
infixr 9 .
(.):: (b -> c) -> (a -> b) -> (a -> c)
(.) f g x = f (g x)
```

```
> (head . tail) [1,2,3]
2
```

```
> ((^ 2).(3 *)) 2
36
```

Iteración de funciones: iterate

```
-- iterate f x = [x,f x, f (f x),...]
iterate:: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
> iterate (* 2) 1
[1,2,4,8,...]
```

```
> take 3 (map head (iterate tail [1..]))
[1,2,3]
```

zipWith , zip

```
-- zipWith f xs ys:  combina los elementos de xs e ys  mediante f
-- Si una lista es más corta, se descartan los sobrantes de la otra
-- zipWith f [x1,...,xn] [y1,...,ym] = [f x1 y1,...,f xk yk]
--    siendo k=min(n,m)

zipWith ::  (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b :  zipWith f as bs
zipWith _ _      _      = []

-- zip xs ys:  combina en parejas los elementos de xs e ys
zip = zipWith hazpar
        where hazpar x y = (x,y)
```

where, *let* permiten definir funciones locales, no solo valores

```
> zip [1,2,3] ['a','b']
 [(1,'a'),(2,'b')]
```

```
> zipWith (+) [1,2,3] [4,5,6]
 [5,7,9]
```

La familia *fold*

Un patrón de recursión sobre listas muy repetido

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

```
and [] = True
and (x:xs) = x && and xs
```

```
or [] = False
or (x:xs) = x || or xs
```

```
length [] = 0
length (x:xs) = 1+length xs
```

```
concat [] = []
concat (xs:xss) = xs++concat xss
```

El patrón general que reconocemos es:

```
g  []       = e
g  (x:xs)   = f x (g xs)
```

O bien, si f viene como un operador infijo $\bigoplus$:

$$
\begin{aligned}
&g\ \ []\qquad &= e\\
&g\ \ (x:xs)\qquad &= x \bigoplus g\ xs
\end{aligned}
$$

O, en términos semiformales, y suponiendo que $\bigoplus$ asocia a la derecha:

$$g\ [x1,x2,\ldots,xn] = x1 \bigoplus x2 \bigoplus \ldots \bigoplus xn \bigoplus e$$

Podemos abstraer en una función de OS con $f/\bigoplus$ y e como parámetros

La familia *fold* (II)

foldr f e [x1,x2,...,xn] = f x1 (f x2 (. . . (f xn e). . .))
foldr $\bigoplus$ e [x1,x2,...,xn] = x1 $\bigoplus$ x2 $\bigoplus$ . . . $\bigoplus$ xn $\bigoplus$ e

```
foldr::  (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
-- O escribiendo f en modo infijo
foldr f e (x:xs) = x `f` (foldr f e xs)
```

```
sum     = foldr (+) 0
product = foldr (*) 1
and     = foldr (&&) True
or      = foldr (||) False
length  = foldr f 0  where f x y = y+1
concat  = foldr (++) []
```

La familia *fold* (III)

foldl $\bigoplus$ e [x1,x2,...,xn] = e $\bigoplus$ x1 $\bigoplus$ x2 $\bigoplus$ ... $\bigoplus$ xn

(suponiendo que $\bigoplus$ asocia por la izda)

foldl f e [x1,x2,...,xn] = (f (...(f (f e x1) x2)...) xn

```
foldl::  (a -> b -> a) -> a -> [b] -> a
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

```
sum     = foldl (+) 0
product = foldl (*) 1
and     = foldl (&&) True
or      = foldl (||) False
length  = foldl f 0
    where f x y = x+1       -- no es la misma f de antes
concat  = foldl (++) []     -- más ineficiente que antes
                            -- Por qué?
```

La familia *fold* (III)

- *foldr* puede procesar listas (incluso infinitas) sin recorrerlas enteras (dependiendo de $\bigoplus$)
- *foldl* ha de recorrer la lista entera
- *foldl* presenta recursión final (*tail recursion)*
- La eficiencia comparativa es muy dependiente de cada caso

Miscelánea OS

```
-- flip f:  cambia de orden los argumentos de f
flip::  (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x

-- curry f:  currifica una función sobre parejas
-- uncurry f:  lo contrario
curry::((a,b) -> c) -> (a -> b -> c)
curry f x y  = f (x,y)

uncurry::(a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y

-- aplicación como operador 'visible'
infixr 0 $
($) ::  (a -> b) -> a -> b
f $ x = f x
```

Funciones que suelen usarse en forma de aplicación parcial

```
-- id:  función identidad
id::  a -> a
id x = x

-- const x:  función constante de valor x
const::  a -> b -> a
const x y = x
```

Funciones anónimas: λ-expresiones

$\lambda x.e \equiv$ función que aplicada a x devuelve e

- $\lambda x.e$ es una función sin nombre (λ-*abstracción*)
- x variable ligada en $\lambda x.e$. La ligadura acaba con e.
- Regla de evaluación (β-*reducción*): $(\lambda x.e)e' = e[x/e']$ donde $e[x/e']$ indica sustitución de x por e' en e.
- $(\lambda x.x+1)\ 2 = 2+1 = 3$ $(\lambda y.y+1)\ 2 = 2+1 = 3$
  $\lambda x.x+1 \equiv \lambda y.y+1$ (α-conversión) $(\lambda x.3)\ 2 = 3$
  $(\lambda x.x+1)((\lambda x.2*x)\ 3) = 7$ ámbito de la ligadura
  $\lambda x.\lambda y.e \equiv \lambda x.(\lambda y.e) \rightsquigarrow$ función de x, y (currificada)
  $(\lambda x.\lambda y.x+y)\ 3\ 2 = (\lambda y.3+y)\ 2 = 3+2 = 5$
  $(\lambda x.\lambda x.x)\ 3\ 2 = ???$

λ-clculo soporta currificación, OS, aplicaciones parciales

- $(\lambda x.\lambda y.x + y)\ 3 = (\lambda y.3 + y)$
- $\lambda f.\lambda g.\lambda x.f(g\ x)$ *composición de funciones*

  $(\lambda f.\lambda g.\lambda x.f(g\ x))\ (\lambda x.2 * x)\ (\lambda x.x + 1)\ 3 = ???$

λ-cálculo (Church,1936)

- Cálculo formal para reflejar cómputos mecanizables
- De potencia equivalente a las máquinas de Turing
- El material base son las λ-expresiones
- Cálculo original muy simple: sin tipos, sin valores primitivos, solo variables, λ's y aplicaciones.
- Gran variedad de variantes y extensiones
- Muy vigente en las ciencias de la computación

λ-expresiones en Haskell

$\lambda x.e$ se escribe `\x -> e` no confundir con la `->` de los tipos

$\lambda x.\lambda y.e$ se escribe `\x y -> e` $\equiv$ `\x->(\y -> e)`

- `\x y -> e` azúcar para `\x -> \y -> e`
- $[\![$`(\x->x+1) 2`$]\!]$ `= 3` `\x->x+1` $\equiv$ `\y->y+1`

  $[\![$`(\x->x+1)((\x->2*x) 3)`$]\!]$ `= 7` $[\![$`(\x y->x+y) 3 2`$]\!]$ `= 5`

  $[\![$`(\x x->x) 3 2 `$]\!]$`= ???` $[\![$`(\x -> \x->x) 3 2`$]\!]$ `= ???`
- Se integra bien con los tipos:

  `\x->x+1::Int->Int` `\x.x::∀a.a->a`

Más ejemplos de uso

$[\![$`map (\x->x+1) [1,2,3]`$]\!]$ `= [2,3,4]`

$[\![$`filter (\x -> x^2<7) [1,3,-2,-3]`$]\!]$ `= [1,-2]`

```
id = \x -> x
```

```
const x = \_ -> x
```

```
incList = map (\x -> x+1)
```

```
zip = zipWith (\x y-> (x,y))
```

λ-expresiones en Haskell soportan ajuste de patrones

$[\![$`(\(x,y) -> x) (3,4)`$]\!]$ `= 3` `--` expresa `fst (3,4)`

$[\![$`(\(x:xs) -> x) [1,2]`$]\!]$ `=` 1 `--` expresa `head []`

$[\![$`(\(x:xs) -> x) []`$]\!]$ `=` $\bot$

Distinción de casos de ajuste vía *case*

```
\x y ->   case x of
              True    -> True
              _       -> y
```

`--` expresa `||`

Un tipo más de expresiones: expresiones *case*

$$
\begin{array}{lll}
\texttt{case} & e \;\texttt{of} & \\
 & t_1 & \texttt{->}\; e_1 \\
 & \ldots & \texttt{->}\; \ldots \\
 & t_n & \texttt{->}\; e_n
\end{array}
$$

- $t_1, \ldots, t_n$: patrones lineales
- $e, e_1, \ldots, e_n$: expresiones
- Se evalúa e y se ajusta con $t_1, \ldots$ hasta que ajuste uno (si no, error)
- Regla de indentación para t_i->e_i

Listas intensionales (*comprehension lists*)

Matemáticas: conjuntos por comprensión

- $\{x^2 \mid x \in \{1, \ldots, 5\}\}$ $\{1, 4, 9, 16, 25\}$
- $\{x + 1 \mid x \in \{1, \ldots, 10\}, x \text{ es primo}\}$ $\{3, 4, 6, 8\}$

Haskell: map, filter, concat

- `map (\x -> x^2) [1..5]`
- `map (\x -> x+1) (filter primo [1..10])`

O también: *listas intensionales*

- `[x^2 | x <- [1..5]]`
- `[x+1 | x <- [1..10] , primo x]`
- `primo x = [y | y <-[2..x-1], mod x y == 0] == []`

Morfología de las listas intensionales

- `[ x+1 | x <- [1..10] , primo x]` expresión principal
- `[x+1 | x <- [1..10] , primo x]` generador
- `[x+1 | x <- [1..10] , primo x]` filtro

En general, una lista intensional es: `[e | c`$_1$`,...,c`$_n$`]`

- e es una expresión
- c$_1$ es un generador
- Cada c$_i$ con $i > 1$ es un generador o un filtro
- Un generador es de la forma `p <- l`
  - `l` es una expresión de tipo $[\tau]$
  - `p` es un patrón de tipo τ
- Un filtro es una expresión booleana

Generadores

- Puede haber varios generadores seguidos
  `[(x,y) | x <- [1,2] , y <- [A,B,C]]`
  `[(1,A),(1,B),(1,C),(2,A),(2,B),(2,C)]`
- El orden influye
  `[(x,y) | y <- [A,B,C] , x <- [1,2]]`
  `[(1,A),(2,A),(1,B),(2,B),(1,C),(2,C)]`

  Generadores múltiples actúan al modo de bucles anidados
- Un generador `p <-l` con `p` no variable tiene un efecto filtro por el ajuste de patrones
  `[x+y|((x,y),A) <- [((2,1),A),((1,3),B),((2,4),A)]]`
  `[3,6]`

Filtros

- Cada filtro actúa sobre el resultado de los generadores y filtros anteriores
- Si un filtro no produce ningún resultado con éxito, la lista queda vacía

Vinculación de variables y listas intensionales

- Variables vinculadas fuera de la lista pueden usarse en cualquier sitio de la lista. La expresión principal puede usar además variables vinculadas dentro de la lista
  `let u=3 in [u+x | x <- [1..u]]`
- Cada generador `p <- t` vincula las variables de `p`, que pueden usarse en los generadores o filtros posteriores, así como en la expresión principal
  `f n = [(x,y) | x <- [1..n] , y <- [1..x]]`
- El ámbito de una variable vinculada en una lista intensional termina con la lista

Un ejemplo elegante: *quicksort*

```
qsort          ::  Ord a => [a] -> [a]
qsort  []      =   []
qsort  (x:xs)  =   qsort [u | u <- xs, u < x]
                   ++ [x] ++
                   qsort [u | u <- xs, u > x]
```

Ternas pitagóricas: (x, y, z) tales que $x^2 + y^2 = z^2$

```
-- ternasP n = ternas pitagóricas con números <= n
ternasP n = [(x,y,z)| x <- [1..n],
                      y <- [1..n],
                      z <- [1..n],
                      z^2 == x^2+y^2]
```

⟦ternasP 10⟧ = [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]

```
ternasP' n = [(x,y,z)| z <- [1..n],
                       x <- [1..z-1],
                       y <- [1..x-1],
                       z^2 == x^2+y^2]
```

⟦ternasP' 10⟧ = [(3,4,5),(6,8,10)]

```
-- ternasP'' = ternas pitagóricas, sin límites
ternasP'' = [(x,y,z)| z <- [1..],
                      x <- [1..z-1],
                      y <- [1..x-1],
                      z^2 == x^2+y^2]
```

Map, *filter*, *concat* y listas intensionales

- `map f xs = [f x | x <- xs]`
- `filter p xs = [x | x <- xs , p x]`
- `concat xxs = [x | xs <- xxs , x <- xs]`
- Recíprocamente toda lista intensional puede expresarse en términos de `map`, `filter` y `concat`.

☞ Las listas intensionales son azúcar sintáctico

```
[x^2 | x <- [1..5]]
```

```
map (\x -> x^2) [1..5]
```

O bien

```
map f [1..5] where f x = x^2
```

En general: `[e | x <- l]` ≡ `map f l where f x = e`

```
[x+1 | x <- [1..10] , primo x]
```

```
map f (filter primo [1..10]) where f x = x+1
```

O bien, más sistemático

```
[x+1 | x <- filter p [1..10]] where p x = primo x
≡ map f (filter p [1..10]) where  p x = primo x
                                  f x = x+1
```

En general:

`[e | x <- l, b]` ≡ `[e | x <- filter p l] where p x = b`

[(x,y)| x <- [1,2] , y <- [3,4]]

`[e| x<-l, y<-l',...]` $\equiv$ `concat [[e| y<-l',...]  | x<-l]`

En el ejemplo:

```
[(x,y)| x<-[1,2] , y <-[3,4]]
≡ concat [[(x,y) | y <-[3,4]] | x<-[1,2]]
≡ concat (map f [1,2]) where f x = [(x,y) | y<-[3,4]]
≡ concat (map f [1,2]) where f x = map (\y->(x,y)) [3,4]
```

Inferencia de tipos

No tenemos en cuenta clases de tipos

- Tipos simples

$TS \ni \tau ::=$		α	*variables de tipo*
	$\mid$	b	*tipos básicos Bool, Int,...*
	$\mid$	$(\tau_1, \ldots, \tau_n)$	*tuplas*
	$\mid$	$T\ \tau_1\ \ldots\ \tau_n$	*Tipo construido*$^{(*)}$
	$\mid$	$\tau_1 \to \tau_2$	*funciones*

  (*): Incluye el caso de las listas: $[\tau]$

- Tipos genéricos (o 'esquemas de tipo')
  $TG \ni \sigma ::= \tau \mid \forall\alpha.\sigma$
  - Es decir: $\sigma \equiv \forall\alpha_1 \ldots \forall\alpha_n.\tau$
  - σ es *cerrado* si todas las variables de τ están en $\forall\alpha_1 \ldots \forall\alpha_n$
  - Si $\sigma \equiv \forall\alpha_1 \ldots \forall\alpha_n.\tau$, al tipo simple τ (posiblemente con un renombramiento de variables) le llamamos *instancia genérica* de σ.

Inferencia de tipos (II)

- Objetivo: dado un programa que define unas funciones $f_1, \ldots, f_n$, inferir los tipos genéricos más generales (*tipos principales*) para $f_1, \ldots, f_n$ que sean compatibles con sus definiciones (o descubrir que el programa está mal tipado).
- Suponemos ordenadas $f_1, \ldots, f_n$ de modo que la definición de cada f_i dependa solo de las anteriores $^{(*)}$. Inferimos los tipos en ese orden, de modo que al inferir el tipo de f_i ya disponemos del tipo de las funciones de las que depende.
  (*): Más en general, particionamos $f_1, \ldots, f_n$ en bloques de funciones $B_1, \ldots, B_k$ ordenados de modo que:
    - Las funciones de cada bloque B_i dependen mutuamente unas de otras (son mutuamente recursivas)
    - Cada bloque B_i depende solo de los bloques anteriores
    - Inferimos los tipos bloque a bloque

Inferencia de tipos (III)

Para cada función f (en general, para cada bloque), la inferencia de tipo se puede descomponer en las siguientes **fases**:

- **Decoración** con tipos de las reglas que definen f
- **Generación de restricciones** de tipo, que son ecuaciones entre tipos que definen lo que se llama un *problema de unificación*.
- **Resolución de las restricciones** de tipo (o sea, del problema de unificación).
- **Generalización** del tipo obtenido.

Decoración con tipos

Las constructoras de datos y algunos símbolos de función tienen ya tipos establecidos (o inferidos previamente), que denominamos *suposiciones de tipo (type assumptions)* para la inferencia en curso. Deben ser esquemas de tipo cerrados. Por ejemplo:

True:: Bool	*0::Int*
$[]{::}\forall\alpha.[\alpha]$	$(:){::}\forall\alpha.\alpha \rightarrow [\alpha] \rightarrow [\alpha]$
$(\&\&){::}Bool \rightarrow Bool \rightarrow Bool$	$(.){::}\forall\alpha\beta\gamma.(\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \beta$

Decoración con tipos (II)

Todas las reglas $f\ t_1 \dots t_n = e$ se decoran con tipos simples del siguiente modo:

- Cada símbolo se decora con:
  - una variable de tipo α *'fresca'* si el símbolo no tiene suposición previa.
  - una instancia genérica fresca de su tipo, en otro caso.
- Cada aplicación $(e\ e_1 \dots e_n)$ se decora como $(e :: \tau\ e_1 :: \tau_1 \dots e_n :: \tau_n) :: \alpha$, donde $\tau, \tau_1, \dots, \tau_n$ son las decoraciones de $e, e_1, \dots, e_n$.
- Cada tupla $(e1, \dots, e_n)$ se decora como $(e_1 :: \tau_1, \dots, e_n :: \tau_n) :: \alpha$, donde $\tau_1, \dots, \tau_n$ son las decoraciones de $e_1, \dots, e_n$
- Cada λ-abstracción $\lambda x.e$ se decora como $(\lambda x :: \alpha.e :: \tau) :: \beta$, donde τ es la decoración de e.

Las variables de tipo introducidas $\alpha, \beta, \dots$ deben contemplarse como 'incógnitas' cuyo valor es descubierto en las siguientes fases.

Decoración con tipos: condiciones adicionales

- Al inferir el tipo de una función f todas las apariciones de f en las reglas que la definen han de decorarse con la misma variable de tipo. Esto se generaliza al caso de inferencia de bloques de funciones mutuamente recursivas.
- Todas las apariciones de las variables de un parámetro formal en su ámbito léxico (una misma regla, una misma λ-abstracción) han de decorarse con la misma variable de tipo. Pero si se repiten en otro ámbito (otra regla, otra λ-abstracción), se decoran con otra variable.
- Distintas apariciones de un símbolo con suposición de tipo (incluso en una misma regla) se decoran con instancias genéricas frescas.

Generación de restricciones de tipo (ecuaciones entre tipos)

A partir de las reglas (y sus subexpresiones) decoradas generamos una colección de ecuaciones entre tipos simples:

- Cada regla decorada $e :: \tau = e' :: \tau'$ genera la ecuación $\tau = \tau'$
- Cada aplicación $(e :: \tau \ e_1 :: \tau_1 \ldots e_n :: \tau_n) :: \tau'$ genera $\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau'$
- Cada tupla $(e_1 :: \tau_1, \ldots, e_n :: \tau_n) :: \tau$ genera $\tau = (\tau_1, \ldots, \tau_n)$
- Cada λ-abstracción $(\lambda x :: \tau.e :: \tau') :: \tau''$ genera $\tau'' = \tau \rightarrow \tau'$

Resolución de las ecuaciones de tipos

- La colección de ecuaciones obtenida define lo que se denomina un *problema de unificación sintáctica*.

- Resolverlo consiste en hallar valores de las variables que conviertan a todas las ecuaciones en identidades sintácticas. Por ejemplo, la solución de $(\alpha, Bool) = (\beta, \alpha)$ vendrá dada por $\alpha = Bool, \beta = Bool$. A cada solución se le llama *unificador*.

- En general, puede haber más de un unificador. Por ejemplo, un unificador para $\alpha = \beta \to \gamma, [\beta] = [\gamma]$ viene dado por $\alpha = Int \to Int, \beta = Int, \gamma = Int$ y otro por $\alpha = Bool \to Bool, \beta = Bool, \gamma = Bool$. Nos interesa el *unificador más general (umg)*, que en el ejemplo es $\alpha = \beta \to \beta, \gamma = \beta$. Se demuestra que, si hay unificadores, hay con seguridad un umg.

- Hay muchos algoritmos para obtener umg's. Aquí presentamos el de *Martelli-Montanari*, que consiste en un proceso de transformación paso a paso del conjunto de ecuaciones hasta llegar a una *forma resuelta*.

Algoritmo de Martelli-Montanari

Dado un conjunto de ecuaciones $\tau_1 = \tau'_1, \dots, \tau_n = \tau'_n$, aplicar reiteradamente y mientras se pueda las siguientes reglas (no importa el orden en el que se consideran las ecuaciones ni las reglas del algoritmo). La notación $E \vdash E'$ indica que el conjunto de ecuaciones E se transforma en E'. *Fallo* indica que no hay unificador.

- *(Eliminación)* $\tau = \tau, E \vdash E$
- *(Descomposición)*
  $\tau_1 \to \tau_2 = \tau'_1 \to \tau'_2, E \vdash \tau_1 = \tau'_1, \tau_2 = \tau'_2, E$
  $(\tau_1, \dots, \tau_n) = (\tau'_1, \dots, \tau'_n), E \vdash \tau_1 = \tau'_1, \dots, \tau_n = \tau'_n, E$
  $T\ \tau_1 \dots \tau_n = T\ \tau'_1 \dots \tau'_n, E \vdash \tau_1 = \tau'_1, \dots, \tau_n = \tau'_n, E$
- *(Ligadura)* $\alpha = \tau, E \vdash \alpha = \tau, E'$, siendo E' el resultado de sustituir α por τ en E, y suponiendo que α aparece en E, pero no en τ.
- *(Reorden)* $\tau = \alpha, E \vdash \alpha = \tau, E$, si τ no es una variable.
- *(Conflicto)* $\tau = \tau', E \vdash \mathit{Fallo}$, si ni τ ni τ' son variables y no son tipos con estructuras homólogas.
- *(Occur-check)* $\alpha = \tau, E \vdash \mathit{Fallo}$, si $\alpha \neq \tau$ pero α aparece en τ

Generalización del tipo

Esta fase consiste simplemente en hacer el cierre universal del tipo simple obtenido en la fase anterior. O sea, si el tipo simple para f dado por el unificador es τ y $\alpha_1, \ldots, \alpha_n$ son las variables de τ, entonces el esquema de tipo inferido finalmente para f es

$$\forall\alpha_1 \ldots \forall\alpha_n.\tau$$

Tipos construidos (definidos por constructoras de datos)

Un caso particular: tipos enumerados

data $T = C_1 \mid \ldots \mid C_n$

data DiaSemana = L | M | X | J | V | S | D
ayer:: DiaSemana $\rightarrow$ DiaSemana
ayer L = D
ayer M = L
. . .
ayer D = S

data Palo = Oros | Copas | Espadas | Bastos

Algunos tipos predefinidos se ajustan a este modelo

data Bool = True | False
data Char = 'A' | . . . 'Z' | 'a' | . . . |'Z'
data Int = -2147483648 | . . . -1 | 0 | 1 | . . . | 2147483647

Forma general de un tipo de datos construido

data $T\ \alpha_1 \ldots \alpha_n = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1} \ldots \tau_{mk_m}$

- T es un identificador (constructora de tipos): aumentan la sintaxis de los tipos simples
- Cada C_i es un identificador (constructora de datos para el tipo T)
- Las constructoras de datos han de ser distintas para cada tipo.
- $\alpha_1 \ldots \alpha_n$ son variables de tipo (parámetros formales del tipo T). En el lado derecho de la definición *data* deben aparecer todas ellas (y ninguna otra)
- Puede ser $n = 0$ (T es un tipo no parametrizado o monomórfico) o $n > 0$ (T es un tipo parametrizado o polimórfico)
- Para cada $i = 1, \ldots, m$ puede ser $k_i = 0$ (C_i es una constante de datos) o $k_i > 0$ (C_i es una constructora de datos no constante)

Un tipo monomórfico: cartas de la baraja

- `data Carta = Carta Int Palo`
- Esta definición determina el tipo de las constructoras de datos: $Carta::Int \rightarrow Palo \rightarrow Carta$
- Algunos valores de *Carta*: *Carta 1 Oros* , *Carta 22 Copas*

O también

```
data Valor = As | Dos | Tres | ...| Sota | Caballo | Rey
data Carta = Carta Valor Palo
```

Tipos recursivos y/o polimórficos

Números naturales al estilo Peano

- `data Nat = Cero | Suc Nat`
- Esta definición determina el tipo de las constructoras de datos:
  Cero::Nat *Suc::Nat* $\rightarrow$ *Nat*
- Algunos valores del tipo Nat: *Cero*, *Suc Cero*, *Suc (Suc Cero)*

Listas polimórficas

- `data List a = Nil | Cons a (List a)`
- *Nil::List a* *Cons:: a* $\rightarrow$ *List a* $\rightarrow$ *List a* $\forall a$ implícito
- Esta definición es isomorfa a la de las listas predefinidas
- *List Int* $\ni$ *Cons 5 (Cons 2 (Cons 1 Nil))* $\simeq [5, 2, 1]$
  List (List Bool) $\ni$ *Cons Nil (Cons (Cons True Nil) Nil)*
  $\simeq [[], [True]]$

Árboles binarios

- Con información solo en las hojas
  data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)
- Con información en hojas y nodos
  data Arbol' a b = Hoja' a | Nodo' b (Arbol' a b) (Arbol' a b)

Tipos *Maybe* y *Either* (en Prelude)

- data Maybe a = Nothing | Just a
- data Either a b = Left a | Right b

Alias de tipo (o tipos sinónimos)

- type String = [Char] está en `Prelude`
- type Coordenada = Float
  type Punto = (Coordenada,Coordenada)
  distancia:: Punto -> Punto -> Float
  distancia (x,y) (x',y') = sqrt ((x-x')^2 + (y-y')^2)
- ```
  > :t distancia
  Punto -> Punto -> Float
  ```
  ```
  > :t (2,3)::Punto
  Punto
  ```
  ```
  > ((2,3)::Punto) == ((2,3)::(Float,Float))
  True
  ```
- No pueden ser recursivos
  type T = (Int,[T]) Error!
- Pero sí pueden ser paramétricos
  type Terna a = (a,a,a)
  ```

Clases de tipos: polimorfismo *ad-hoc*

Polimorfismo paramétrico (sistema de *Hindley-Milner*)

length :: $\forall a.[a] \rightarrow Int$

- *length* se puede aplicar a listas de *cualquier* tipo
- La definición de *length* es uniforme para todos los tipos

¿Qué tipo queremos que tengan $+$, $*$, $==$, $\ldots$?

Polimorfismo *ad-hoc*

$(+)$ :: $\forall a \in Num.a \rightarrow a \rightarrow a$

$(==)$ :: $\forall a \in Eq.a \rightarrow a \rightarrow Bool$

```
(+)::Num a ⇒ a → a → a
(==)::Eq a ⇒ a → a → Bool
```

Tipos cualificados

- *Num* es una *clase* de tipos
- La definición de $+$ puede ser distinta para cada tipo de *Num*

Las clases de tipos fueron propuestas por *Wadler*

Clase de tipo = colección de tipos + métodos de clase

- Al definir una clase $\mathcal{C}$ se introducen sus métodos, pero no qué tipos están en $\mathcal{C}$.
- Los tipos de $\mathcal{C}$ se van introduciendo por declaraciones de *instancia* de $\mathcal{C}$.
- La definición de una clase $\mathcal{C}$
  - *Debe* incluir la declaración de sus métodos
  - *Puede* incluir la definición por defecto de sus métodos
  - Al declarar un tipo como instancia de $\mathcal{C}$ se puede cambiar la definición por defecto de los métodos
- Una vez definida $\mathcal{C}$ se pueden definir funciones con tipos cualificados por $\mathcal{C}$

Definición de clases de tipos

Ejemplo: la clase *Eq*

```
class Eq a where
  (==),(/=) ::  a -> a -> Bool   -- Métodos de la clase Eq
   x == y = not (x/=y)          -- Definiciones por defecto
   x /= y = not (x==y)          -- de los métodos
```

- Los tipos de los métodos quedan cualificados:
  `(==),(/=)::  Eq a => a -> a -> Bool`
- Las definiciones por defecto de los métodos son opcionales
- En una instancia de `Eq` bastará redefinir `==` o bien `/=` .
  (Tanto $\{==\}$ como $\{/=\}$ son *conjuntos minimales suficientes* de métodos de Eq)

☞ La clase `Eq` ya está definida en `Prelude`

Ahora podemos definir funciones que usen métodos de *Eq*

- ```
  elem x [] = False
  elem x (y:ys) = if x==y then True else elem x ys
  ```

  ¿Tipo de *elem*?

  ```
  elem::  Eq a => a -> [a] -> Bool
  ```

- No hace falta tener definidas instancias de la clase para definir funciones que usen los métodos de la clase.
  ```

Declaración de instancias de clase

Declaración de *Bool* como instancia de *Eq*

```
data Bool = False | True

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

Hemos optado por redefinir ==

Al declarar un tipo T como instancia de una clase $\mathcal{C}$, todos los métodos de $\mathcal{C}$ deben quedar definidos para T, bien por su definición por defecto, si la tienen, o por la (re)definición del usuario

Otras declaraciones (condicionadas) de instancia de *Eq*

```
instance Eq a => Eq [a]  where
  []     == []     = True
  []     == (_:_)  = False
  (_:_)  == []     = False
  (x:xs) == (y:ys) = x==y && xs==ys
```

☞ [a] está en Eq si a está en Eq

```
instance (Eq a,Eq b) => Eq (a,b)   where
  (x,y) == (x',y') = x==x' && y==y'
```

☞ (a,b) está en Eq si a y b están en Eq

Estas declaraciones concretas ya están en el Prelude

Declaraciones automáticas de instancia de clase

Usar *deriving* nos ahorra trabajo

`data Bool = True | False deriving Eq`

`data Arbol a b = Hoja a | Nodo b (Arbol a b) (Arbol a b) deriving Eq`

- Al usar *deriving Eq* al definir un tipo construido T se genera automáticamente la definición de $==$ como igualdad estructural (o *sintáctica*) de los valores de T
- Se puede usar *deriving* para las clases *Eq*,*Ord*,*Enum*,*Show*, entre otras
- No se puede usar *deriving* para clases definidas por el usuario

deriving no es siempre adecuado o posible

```
type Numerador = Integer
type Denominador = Integer
infixl 7 :/
data Fraccion = Numerador :/ Denominador
instance Eq Fraccion where
  a:/b== c:/d = a*d == b*c
instance Num Fraccion where
  a:/b + c:/d = (a*d+b*c) :/ b*d
  a:/b * c:/d = (a*c) :/ b*d
  --- Más operaciones de la clase Num ---
```

Alias de tipo

Constructora de datos infija

$Integer \in Eq$

Clases de tipos: subclases

La clase *Ord* como subclase de *Eq*

- En la clase *Ord* queremos tener a los tipos cuyos valores pueden ser comparados por $<, >, \leq, \geq$.
- Para que $x \leq y$ tenga sentido, ha de tenerlo $x == y$.
- Así pues, todo tipo de *Ord* debe estar también en *Eq*. Podemos decir que *Ord* es subclase de *Eq*.
- Declaramos `class Eq a => Ord a where`

```
-- métodos de Ord --
```

- La comprobación de tipos usa que si $a \in Ord$ entonces forzosamente $a \in Eq$

Pero se escribe `Eq a => Ord a`!

- Pero declarar un tipo como instancia de *Ord* **no** implica tenerlo declarado como instancia de *Eq*. Hay que hacerlo explícitamente.

```haskell
data Ordering = LT | EQ | GT
class Eq a => Ord a where
  infix 4 <,>,<=,>=
  compare ::  a -> a -> Ordering
  (<), (<=), (>=), (>) ::  a -> a -> Bool
  max, min ::  a -> a -> a
  -- Conjunto minimal suficiente:  (<=) 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
  ...
```

Ya incluido en el `Prelude`

Bool como instancia de Ord

```
-- Tenemos data Bool = False | True

instance Ord Bool where
  True <= False = False
  _    <= _     = True
```

O bien: `data Bool= False | True deriving (Eq,Ord)`

Listas polimórficas como instancia de Ord

```
-- Tenemos data [a] = [] | a:[a]

instance Ord a => Ord [a]  where
  [] <= []     = True
  [] <= (_:_)      = True
  (_:_) <= []      = False
  (x:xs) <= (y:ys) = x < y || x == y && xs <= ys
```

Aquí también valdría *deriving*

Orden inducido por *deriving Ord*

Supongamos `data T = C1 ...| C2 ...| Cn ...deriving Ord` , y supongamos dos valores `x` e `y` de `T` .

Para evaluar `x <= y` :

1. Se comparan las constructoras más externas de `x` e `y` , que están ordenadas según el orden de aparición en la def. de `T` .
2. Si son iguales, se comparan lexicográficamente las tuplas de argumentos. Es decir, se comparan primero los primeros argumentos; si son iguales, se pasa al segundo, etc.

El orden inducido por *deriving* no siempre es el adecuado

```
instance Ord Fraccion where
  (a :/ b) <= (c :/ d) = a*d <= b*c
```

¿Qué orden resultaría con `data Fracción = ...deriving Ord` ?

Jerarquía de clases predefinidas en Haskell

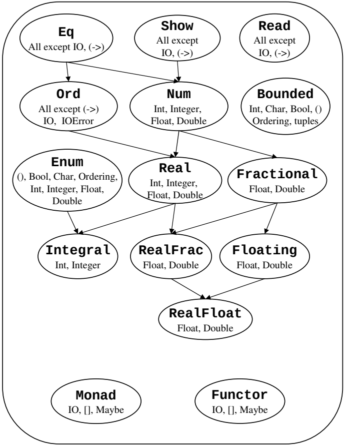


Figure 6.1: Standard Haskell Classes

Extraído del Haskell2010 report

Ambigüedad de tipos

- Es un problema específico del polimorfismo de clases
- Se presenta cuando al evaluar una expresión `e` cuyo tipo incluye una restricción de clase `C a => ...` el análisis de tipos no tiene información suficiente para saber qué instancias particulares deben usarse para evaluar los métodos de clase que puedan intervenir en `e`. En ese caso el sistema nos da un error de *ambigüedad de tipos*, que no indica que la expresión esté mal tipada, sino que se necesita dar información más explícita.

Ejemplo

```
>:t toEnum 0
Enum a => a
```

```
> toEnum 0
Error ambigüedad tipos
```

`toEnum 0` está bien tipada: dado cualquier tipo `a` de la clase `Enum` , `toEnum 0` nos da un valor del tipo `a`. Por ejemplo `toEnum 0` nos da `0` en `Int`, `False` en `Bool`, ... Pero por sí sola `toEnum 0` no tiene información suficiente para saber cómo evaluarla y da un error de ambigüedad. Sin embargo:

- `(toEnum 0)::Int` dará `0`
- `(toEnum 0)::Bool` dará `False`
- ¿Qué da `head [toEnum 0]`?
- ¿Qué da `head [toEnum 0,1]`? ¿Por qué?

Entrada/salida

- La entrada/salida es un problema en el paradigma funcional puro
- En Haskell: entrada/salida *monádica*
- Mónadas: Abstracción adecuada para integrar en el sistema de tipos cómputos efectuados en secuencia que acarrean efectos laterales (no necesariamente I/O)
- Propuestas por Phil Wadler (adaptando ideas previas de otros)

¿Por qué es un problema la E/S para Haskell?

¿Qué pasaría si tuviésemos, por ejemplo, `getInt::Int`?

- Por el principio de que toda expresión e de tipo T tiene un valor $[\![e]\!]$ del tipo T, $[\![\texttt{getInt}]\!]$ sería un número entero.
- Por la transparencia referencial $[\![\texttt{getInt}]\!]$ debería ser único, independientemente de dónde y cuándo se use `getInt`.
- Podemos formar una expresión como `getInt - getInt`.
- Hasta la fecha se tenía: $[\![e - e']\!] = [\![e]\!] - [\![e']\!]$
- Debería entonces ser:
  $[\![\texttt{getInt - getInt}]\!] = [\![\texttt{getInt}]\!]$ - $[\![\texttt{getInt}]\!] = 0$
- Absurdo! Significaría que al evaluar `getInt - getInt` leemos dos veces el mismo entero
- Similar: `getInt + getInt` debería ser equivalente a `2*getInt`
- Más cosas: hasta la fecha, para evaluar $e - e'$ podemos, si queremos, evaluar e y e' en paralelo. Ahora ya no!

Entrada/Salida en Haskell

El tipo IO a

- Procesos de entrada/salida: expresiones del tipo `IO a`
- ¿Qué es un valor del tipo `IO a`?: *acción* de I/O que, **si se efectúa**, produce un efecto de I/O con un resultado asociado de tipo `a`.
- `getInt::IO Int`: acción de leer un entero
- `getChar::IO Char`: acción de leer un carácter
- `putChar::Char -> IO ()`: acción de escribir un carácter

  `()` $\rightsquigarrow$ tipo con un solo valor, que es también `()`

  Las acciones de tipo `IO ()` no generan valor asociado
- La acción representada por una e`::IO a` se efectúa si e es evaluada al evaluar la expresión escrita en la consola o la expresión `main` de un módulo.

Algunas acciones I/O básicas y predefinidas

- `getChar:: IO Char`
- `putChar:: Char -> IO ()`
- `return:: a -> IO a`
  `return x` $\leadsto$ acción sin efecto lateral con valor asociado `x`
  Hace falta para expresar acciones complejas
- `getLine:: IO String`
- `putStr:: String -> IO ()`
- `print :: Show a => a -> IO ()`
  `print = putStr . show`

Secuenciación de acciones de I/O: operador `>>=` (*'then'*)

```
infixl 1 >>=
(>>=)::  IO a -> (a -> IO b) -> IO b
```

- `IO a` — primera acción; tendrá un valor asociado `x::a`
- `(a -> IO b)` — `x::a` determina la segunda acción
- `IO b` — el resultado es esta segunda acción

Leer una línea y repetirla dos veces

```
eco2::  IO ()
eco2 = getLine >>= \xs -> putStr (xs++"\n"++xs++"\n")
```

putStr,getLine definidas mediante putChar,getChar

```
putStr::  String -> IO ()
putStr []     = return ()
putStr(c:cs) = putChar c >>= \_ -> putStr cs

getLine::IO String
getLine = getChar >>= \c ->
          if c=='\n' then return []
                     else getLine >>= \cs ->
                          return (c:cs)
```

Secuenciación de acciones: notación do

```
do  x1 <- e1        e1::IO a1, x1::a1
    e2              e2::IO a2
    ...             ...
    xn <- en        en::IO an, x::an
    ...             ...
    em              em::IO am ⇝ valor del do
```

do es un azúcar sintáctico

```
do  x <- e      ≡   e >>= \x -> e'
    e'
```

```
do  e           ≡   e >>= \_ -> e'
    e'
```

A `do` se le aplica la regla de indentación

Leer una línea y repetirla dos veces

```
eco2::  IO ()
eco2 = do xs <- getLine
          putStr (xs++"\n"++xs++"\n")
```

putStr,getLine definidas mediante putChar,getChar

```
putStr::  String -> IO ()
putStr []     = return ()
putStr(c:cs) = do putChar c
                  putStr cs

getLine::IO String
getLine = do c <- getChar
             if c=='\n' then return []
                        else do cs <- getLine
                                return (c:cs)
```

Leer un entero

```
getInt::  IO Int
getInt = do line <- getLine
            return (read line::Int)
```

¿por qué hace falta `::Int` ?

`getInt` no es predefinida del Prelude

Lee enteros y escribe sus cubos hasta que el entero sea 0

```
cubos::  IO ()
cubos =
 do putStrLn "Escribe un entero (0 para salir)"
    n <- getInt
    if n==0 then return ()
            else do putStr (show n++"^3 = ")
                    print (n^3)
                    cubos
```

La transparencia referencial, a salvo

- `getInt - getInt` sin sentido, mal tipado
-
```
do  x <- getInt
    y <- getInt
    print (x-y)
```
no tiene por qué ser $= 0$
-
```
do  x <- getInt
    print (x+x)
```
$\simeq$
```
do  x <- getInt
    print (2*x)
```

La clase `Monad`

- `IO` es una instancia de la clase `Monad`
- `return` , `>>=` son métodos de `Monad`
- La notación `do` es aplicable para expresar secuenciación de cómputos en tipos de `Monad`
- `Monad` es una **clase de constructoras de tipo**
  - `IO` no es un tipo, sino una constructora de tipos
  - Es la constructora de tipos `IO` quien pertenece a `Monad` , no los tipos `IO Int` , `IO String` , ...
  - Otras constructoras de tipo de `Monad` son, por ejemplo, `Maybe` o `[]`