
- título: Tutorial de Haskell
- autor: Francisco Nebrera Perdomo
- institución: Universidad de La Laguna
- correo electrónico: freinn@gmail.com
- fecha: Marzo 2015

Tutorial de Haskell

"Sábete, Sancho, que no es un hombre más que otro, si no hace más que otro" - Don Quijote de la Mancha. Capítulo XVIII.

Reacción típica de un programador al ver su primer fragmento de código Haskell:

Asociatividad de los elementos de Haskell

Asocian **a izquierdas:**

- La aplicación de funciones, a consecuencia de la curryficación.
- `<$>` y `<*>` de la clase de tipos `Applicative` (ambos `infixl 4`).

Asocian **a derechas:**

- Los parámetros de las funciones, a consecuencia de la curryficación.
- La aplicación de funciones con `$`.
- La composición de funciones con `.`.
- El constructor (:) (en inglés `Cons`).

Errores comunes

Es fácil deducir cuál es la cabeza de esta lista de listas:

```
ghci> head [[1,2,3],[4,5,6]]
[1,2,3]
```

Pero yo mismo haciendo una traza en papel deduje que la cola de `[[1,2,3],[4,5,6]]` era `[4,5,6]`. Craso error:

```
ghci> tail [[1,2,3],[4,5,6]]
[[4,5,6]]
```

Optimizaciones de Project Euler

Explicar el problema 47 y alguno más.

Ideas sueltas

curryficar ~= fijar

`Data.Numbers.Primes` exporta la función `primeFactors`, la cual está bastante optimizada.

Las funciones pueden ser pasadas a funciones, y las acciones pueden ser pasadas a acciones.

El operador de composición de funciones, `(.)`, crea una función que aplica su argumento derecho y luego pasa el resultado a su argumento izquierdo, así que usamos eso para combinar la aplicación de dos funciones.

exercism

ARN

Queremos resolver el siguiente problema: dada una cadena de ADN, queremos pasarlo a ARN. Para ello hay que cambiar 'C' por 'G', 'G' por 'C', 'A' por 'U' y 'T' por 'A'.

Usar reconocimiento de patrones para este tipo de cosas es mejor que usar guardianes, los guardianes se ejecutan en orden y no dan tanta información al compilador.

Dos opciones:

```
transcribir 'G' = 'C'
transcribir 'C' = 'G'
…
transcribir _ = error "…"
```

O, un poco mejor:

```
transcribir c = case c of
  'G' -> 'C'
  'C' -> 'G'
  …
  _ -> error "…"

aARN :: String -> String
aARN xs = map transcribir xs
  where
    transcribir c = case c of
                      'C' -> 'G'
                      'G' -> 'C'
                      'A' -> 'U'
                      'T' -> 'A'
                      _   -> error "secuencia de ADN inválida"
```

Prefiero la segunda, ya que es más concisa. También es más a bajo nivel, todas las formas de reconocimiento de patrones se traducen a expresiones `case` en el núcleo de GHC.

Razonando con Haskell

La aplicación de funciones tiene la máxima prioridad, 10.

La evaluación perezosa significa: haz sólo lo que te pida un patrón a la izquierda de una ecuación o cualificador (`where` o `let`).

LYAH

Todo lo que puede hacer una función en Haskell es recibir ciertos parámetros y devolver cierto valor.

Todas las funciones en Haskell realmente reciben un único parámetro. Por tanto una función `a -> b -> c` recibe un único parámetro de tipo `a` y devuelve una función `b -> c`, que recibe un parámetro y devuelve `c`. Por tanto, la aplicación parcial de funciones devuelve una función que toma los parámetros que dejamos sin "rellenar". Así que `a -> b -> c` puede ser reescrita como `a -> (b -> c)`.

Las cosas pueden actuar más como computaciones que como cajas: (`IO` y `(->) r`) pueden ser funtores.

`return` no tiene nada que ver con el `return` de otros lenguajes. No hace que la ejecución de una función termine. Simplemente recibe un valor normal y lo pone en un contexto.

- mapear una función sobre una función produce una función.
- mapear una función sobre un `Maybe` produce un `Maybe`.
- mapear una función sobre una lista produce una lista.

Bind es `>>=`, la analogía de la aplicación de funciones en contextos monádicos.

Bind permite hacer algo análogo al reconocimiento de patrones, sin hacerlo.

Lo que realmente hace el operador bind `>>=` es:

1. "extrae" el valor de un contexto monádico.
2. aplica una función a ese valor extraído.
3. lo envuelve de nuevo en un contexto monádico.

Luego el tipo de bind `(>>=)` es:

```
(>>=) :: m a -> (a -> m b) -> m b
```

En una expresión `do`, todo lo que no sea un `let` es un valor monádico.

Por ello, los funtores aplicativos como mucho pueden ser parámetros de funciones usando el estilo aplicativo.

Las mónadas son superiores y nos permiten encadenar computaciones que podrían fallar, y en caso de fallo este fallo se propaga de una a otra. Si todas tienen éxito, simplemente se encadenan de izquierda a derecha.

El operador `(>>)` recibe una mónada de tipo `a` y otra de tipo `b` (**nota importante:** que haya dos variables de tipo no implica que deban enlazarse a tipos distintos, las mónadas de `(>>)` bien pueden ser del mismo tipo), y lo que hace es..TODO.

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

`return` inyecta un valor en una mónada (contenedor).

De hecho, las compresiones de listas son sólo azúcar sintáctico para usar listas como mónadas. Las compresiones de listas y las listas en notación `do` se traducen a usar >>= en computaciones no deterministas.

El filtrado en las compresiones de listas se resume a usar la función guard con esa condición.

Es mucho mejor empezar las funciones por su cabecera, debido al fuerte sistema de tipos de Haskell.

Normalmente se hace `read "846195673" :: Int` ó `read "5232.488647" :: Float` para pasar de un tipo a otro, si usamos `map` es mejor usar una función con cabecera explícita, que le da la información suficiente al compilador acerca de qué tipo queremos:

```
leerInts ::[String] -> [Int]
leerInts = map read
```

Haskell es muy fiel a las matemáticas reales, teóricas. El reconocimiento de patrones es un "binding". Una compresión de listas equivale a un "para todo x" en matemáticas.

Jugar mucho con la idea de que las Strings son listas de Char, String es sinónimo de tipo [Char]

Idea del paradigma:

Lenguajes imperativos:

Se trata de darle al ordenador una serie de pasos que debe seguir hasta llegar a una solución o a la conclusión de que no existe solución a ese problema.

Lenguajes funcionales:

Se le indica al ordenador qué es cada cosa, y por ello las funciones no tienen permitido tener efectos laterales.

Por tanto, no podemos modificar estructuras de datos existentes, sino construir *nuevas* estructuras de datos que de manera "innata" tienen las modificaciones que queríamos hacer ya hechas.

El hecho de que las funciones no puedan cambiar el estado - como por ejemplo, actualizar variables globales - es bueno porque nos ayuda a razonar sobre nuestros programas. Sim embargo, esto crea algunos problemas: Si una función no puede cambiar nada, ¿cómo se supone que nos devolverá el resultado que calculó?

Haskell cuenta con un buen sistema para tratar con funciones que tienen efectos laterales. Se trata de separar la parte pura de nuestro programa de la parte impura (que se ocupa de la E/S, por ejemplo). Las ventajas que brinda esta separación son dos:

- podemos seguir razonando sobre nuestro programa puro
- seguimos aprovechando las virtudes de la pureza - como evaluación perezosa, robustez, uso de composición - mientras nos comunicamos fácilmente con el mundo exterior.

Variables

Lenguajes imperativos:

Variable en programación imperativa: trozo de memoria mutable con un nombre variable en Haskell, simplemente un nombre que usaremos para la sustitución el valor en Haskell es una forma de decir que es algo permanente.

- Variables: asociaciones cambiables entre nombres y valores.
- Se llaman imperativos porque consisten en secuencias de órdenes.
- Asignaciones: asocian a una variable el resultado de una expresión. Cada expresión de orden puede referir a otras variables que pueden haber sido cambiadas por órdenes anteriores. Esto permite que los valores pasen de orden a orden.

- En los lenguajes imperativos, las órdenes pueden cambiar el valor asociado a un nombre por una orden anterior así que cada nombre puede ser y usualmente será asociado a valores diferentes durante la ejecución de un programa.

En lenguajes imperativos, el mismo nombre puede ser asociado a diferentes valores.

Lenguajes funcionales:

Los lenguajes funcionales se basan en llamadas estructuradas a funciones. Un programa funcional es una expresión consistentente en una llamada a una función que llama a otras funciones.

```
\<función1\>(\<función2\>(\<función3\>...)...))
```

Por tanto, cada función recibe valores de y pasa valores a la función llamadora. Esto se conoce como composición o anidamiento de funciones.

En Haskell se definen las variables, no se asignan. Por ello, se hace sólo una vez, y eso no puede cambiar a lo largo de la ejecución.

Los nombres sólo se introducen como los parámetros formales de las funciones… Cuando un parámetro formal se asocia con un valor de parámetro real, luego no hay manera de asocialo a un nuevo valor. No hay concepto de orden que cambie el valor asociado a través de asignación. Por tanto, no hay concepto de secuencia de instrucciones o repetición de órdenes para activar cambios sucesivos a valores asociados con nombres.

En los lenguajes funcionales, un nombre solo se asocia una vez a un valor.

Orden de ejecución:

Lenguajes imperativos:

Es crucial el orden de ejecución porque los valores se pasan de instrucción a instrucción mediante referencias a variables comunes, y una orden puede alterar un valor antes de ser usado por otra orden. Un cambio en el orden de ejecución podría alterar el comportamiento del programa.

En los lenguajes imperativos, el orden de ejecución es fijo.

Lenguajes funcionales:

En los lenguajes funcionales, las llamadas a funciones no pueden cambiar los valores asociados con nombres comunes. Por lo tanto, el orden en el cual se ejecutan las llamadas anidadas a funciones no importa, porque las llamadas a funciones no pueden interactuar unas con otras.

F (A(D), B(D), C(D)), el orden en el cual A(D), B(D) y C(D) se ejecutan no importa porque las funciones A, B y C no pueden cambiar su parámetro real común D.

En los lenguajes funcionales, no hay orden de ejecución necesario.

Por todo lo expuesto más arriba, el orden de ejecución no afecta el resultado final en los lenguajes funcionales. La independencia en el orden de ejecución es una de las mayores fortalezas de los lenguajes funcionales.

Repetición:

Lenguajes imperativos:

Como las órdenes podrían cambiar los valoes asociados a nombres de órdenes anteriores, de modo que no es necesario introducir un nuevo nombre para cada nueva instrucción. Por ello, para realizar muchos comandos muchas veces, no se necesita duplicar las órdenes. En vez de eso, las mismas órdenes se repiten.

En los lenguajes imperativos, valores nuevos pueden ser asociados con el mismo nombre por medio de la repetición de órdenes.

Lenguajes funcionales:

Como no se pueden reusar nombres con valores diferentes, las funciones anidadas se usan para crear nuevas versiones de los nombres para nuevos valores. Como no se puede usar la repetición de órdenes, se usan llamadas recursivas para crear repetidamente nuevas versiones de nombres asociados a nuevos valores. Aquí, una función se llama a sí misma para crear nuevas versiones de sus parámetros formales los cuales estarán "ligados" a nuevos valores reales de parámetros.

Estructuras de datos en lenguajes funcionales:

En los lenguajes imperativos, los elementos de los vectores (arreglos, arrays, matrices) y estructuras (records en Pascal, structs en C/C++) se cambian mediante asignaciones sucesivas. En los lenguajes funcionales, como no hay asignación, las sub-estructuras de las estructuras de datos no pueden ser cambiadas una por una. En lugar de esto, es necesario reescribir la estructura completa con cambios explícitos a la sub-estructura adecuada.

Los lenguajes funcionales proporcionan representación explícita para las estructuras de datos.

En vez de arrays se usan listas, ya que reescribir arrays es computacionalmente muy costoso. Se basan en notación recursiva.

La capacidad de represnetar estructuras de datos enteras tiene ventajas. Por ejemplo, se usan formatos estándar para mostrar, almacenar y modificar estas estructuras de datos.

No existen las estructuras globales en los lenguajes funcionales. No se pueden cambiar las sub-estructuras independientemente. En lugar de ello, las estructuras de datos enteras son pasadas explícitamente como parámetros reales a funciones para cambiar la sub-estructura, y luego devueltas a la función llamadora. Por tanto, las llamadas a funciones en los lenguajes funcionales son más grandes que sus equivalentes en lenguajes imperativos por esos parámetros adicionales. Sin embargo, tiene la ventaja de asegurar que la manipulación de estructuras mediante funciones es simpre explícita en la definición de la función y sus llamadas. Esto hace más fácil seguir el flujo de los datos en los programas.

Funciones como valores:

En muchos lenguajes imperativos, los subprogramas pueden ser pasados como parámetros reales a otros subprogramas pero es raro para un lenguaje imperativo permitir a los subprogramas ser pasados como resultados.

En los lenguajes funcionales, las funciones pueden construir nuevas funciones y pasárselas a otras funciones.

Los lenguajes funcionales permiten a las funciones ser tratadas como valores. Esto da a los lenguajes funcionales un gran poder y flexibilidad.

Lambda-cálculo = aplicación estructurada de funciones.

En lambda-cálculo, si varios órdenes de evaluación diferentes terminan, los resultados serán idénticos. También se ha demostrado que un orden de evaluación particular conduce más a la terminación que cualquier otro. Por tanto, es mejor ejecutar ciertas partes de un programa en un orden y otras partes en otro. En particular, si un lenguaje es independiente del orden de evaluación quizá sea posible ejecutar partes del programa en paralelo.

"IDEAS IMPORTANTES: Razonando con Haskell"

Composición de funciones

El libro da unas restricciones de argumentos y valores de retorno muy buenas. La composición es asociativa a la derecha.

Truco: para entender la notación de composición con poco riesgo de equivocarse, es muy útil hablar de `f . g` como "`f` después de `g`".

Si la solución a un problema consta de varias etapas, podemos definir cada una de ellas como funciones independientes y componer todas para solucionar el problema.

En Haskell, la función composición es `(.)`, veamos un ejemplo.

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

La expresión `f (g (z x))` es equivalente a `(f . g . z) x`. Por tanto, podemos transformar algo un poco lioso, como esto:

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

En una línea de código mucho más clara, casi autodescriptiva:

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Por tanto, lo primero que hace esta nueva función es el valor absoluto, `abs`, y después cambia el signo `negate`,dando como resultado una lista con todos los números iniciales pasados a negativo.

Composición con múltiples parámetros:

Para conseguir esto, tenemos que valernos de la aplicación parcial de funciones. Veámoslo con un ejemplo:

```
sum (replicate 5 (max 6.7 8.9))
```

Puede ser transformada en:

```
(sum . replicate 5) max 6.7 8.9
```

```
sum . replicate 5 $ max 6.7 8.9
```

Si queremos reescribir una expresión con un montón de paréntesis usando composición de funciones, podemos empezar escribiendo la función más interna y sus parámetros. Delante de ella escribimos `$` y componemos todas las funciones que venían antes escribiéndolas sin su último parámetro y poniendo puntos entre ellas. Por ejemplo:

```
replicate 2 (product (map (*3) (zipWith max [1,2] [4,5])))
```

Puede ser reescrito como:

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

Truco: Una expresión con n paréntesis tendrá n-1 operadores de composición `(.)`.

Estilo de funciones con argumento declarado "point-wise":

```
sum :: (Num a) => [a] -> a
sum xs = foldl (+) 0 xs
```

El xs está lo más a la derecha posible a los dos lados del signo igual. A causa de la currificación, podemos omitir xs en ambos lados, ya que `fold (+) 0` crea una función que recibe una lista. De este modo, estamos creando una función de orden superior.

Estilo de funciones sin argumento declarado "point-free":

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

Ejemplo de paso de un estilo al otro

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

Se convierte en:

```
fn = ceiling . negate . tan . cos . max 50
```

En definitiva, la composición sirve de pegamento entre funciones simples para crear funciones más complejas. Esto es en esencia positivo pues consigue un código legible y fácilmente entendible, pero no se debe abusar de ello creando cadenas de composición demasiado largas, pues al final se podría perder legibilidad y claridad.

Sistema de tipos

Gracias al sistema de tipos, se puede inferir el tipo más general de una función a partir de sus ecuaciones.

Todas las apariciones de una variable de tipos deben ser reemplazadas por el mismo tipo. Si tenemos una función con una variable de tipos a, e intentamos que esa a se corresponda con dos tipos diferentes, obtendremos un error.

Un tipo polimórfico tiene muchos tipos. Un tipo polimórfico es una plantilla (un esquema de tipos) que puede ser usada para crear tipos específicos.

Una expresión en la que intervienen funciones polimórficas es correcta desde el punto de vista de su tipo si se pueden encontrar sustituciones consistentes para las variables de tipo.

Capítulo 4:

Los sinónimos de tipo (y por tanto, también los identificadores de tipo) comienzan con mayúsculas en Haskell.

Capítulo 6:

El constructor (:) es asociativo a la derecha

Capítulo 7:

El anidamiento de varias concatenaciones a la izquierda tiene complejidad cuadrática. El anidamiento de varias concatenaciones a la derecha tiene complejidad lineal.

LYAH:

La flecha de las definiciones del tipo de las funciones es asociativa a la derecha.

Cuando tengamos una declaración de tipos de función con la flecha '->', eso significa que es una función que recibe aquello a la izquierda de la flecha y devuelve un valor cuyo tipo se indica en el lado derecho de la flecha.

Cuando tenemos algo como a -> (a -> a) en realidad se trata de una función que recibiendo un parámetro, nos devuelve otra función que recibe otro parámetro de tipo a, y al hacer su cálculo devuelve otro también del tipo a.

¿Cómo nos beneficia esto a nosotros? Si llamamos a una función pasándole menos parámetros de los que acepta, obtendremos una función parcialmente aplicada, la cual es una función que recibe tantos parámetros como dejamos sin "rellenar". Por tanto, este es un buen método para crear funciones "al vuelo", y después podemos pasárselas a otras funciones.

Primeros pasos

Para empezar a programar en Haskell tenemos varias opciones, ya que Haskell se puede interpretar o compilar.

Lo primero que debemos hacer es instalar un compilador. Para ello tenemos diversas opciones, aunque en el tutorial usaremos el más común y oficial, GHC. En el momento de la escritura de este tutorial, se ha usado una versión beta de GHC 7.10.1, el cual en el momento en el que escribo estas palabras está en RC3. La versión final está al caer y el código de este tutorial ha sido adaptado a esta versión (por ejemplo las instancias de la clase `Monad` deben ser instancias de `Applicative` y por ello, también de `Functor`).

Mi sistema favorito para programar en Haskell es GNU/Linux, y en concreto uso Kubuntu y Manjaro en la actualidad.

Por ello, las instrucciones de instalación de GHC que daré serán válidas en distros derivadas de Debian y de Arch Linux.

Instalación en Arch Linux/Manjaro

Las instrucciones para la instalación de todos los paquetes se pueden encontrar aquí.

Lo primero que debemos hacer es sincronizar la base de datos de los repositorios si no lo has hecho recientemente:

```
sudo pacman -Syyu
```

Si esto falla, podemos intentar la solución típica:

```
sudo pacman-key --init
sudo pacman-key --populate archlinux
sudo pacman-key --populate manjaro
sudo pacman-key --refresh-keys
sudo pacman -Syyu
```

Ahora procedemos a instalar los paquetes más importantes, de los repositorios oficiales:

```
sudo pacman -S ghc cabal-install
```

Esto instala lo más importante, que es ghc (compilador), ghci (intérprete), runhaskell (permite ejecutar código "al vuelo" y hacer scripts de bash), entre otras cosas. Además instala cabal, que es un gestor de paquetes que nos permite instalar las librerías que queramos de los repositorios oficiales de Haskell, las cuales están muy bien optimizadas y suelen ser cómodas de usar.

Instalación en Debian/Ubuntu

Es una instalación muy complicada, mejor llama a un hamijo juanker:

```
sudo apt-get install haskell-platform
```

Después de darle los 30€ que tu amigo se ha ganado, ya puedes limpiarte el sudor de la frente y empezar a programar.

Jugando con GHCi

Para iniciar GHCi usaremos el comando `ghci` en la consola que tengamos, lo cual cargará el intérprete y podremos empezar a usarlo. Podemos compilar ficheros de texto o introducir al vuelo nuestras funciones creadas y evaluarlas. Veamos una sesión de ejemplo para ver qué se puede evaluar sin definir nada previamente:

```
*Main> sqrt 56
7.483314773547883
*Main> sqrt 45.798
6.767421961131137
*Main> 447 + 789798
790245
*Main> "hola"
"hola"
*Main> "ola" ++ " k " ++ "ase"
"ola k ase"
*Main> 2^100
1267650600228229401496703205376
*Main> 2^1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275
```

Como vemos, Haskell soporta aritmética con enteros enormes (tanto como nuestra memoria principal nos permita), y hemos visto ejemplos de operaciones con números y cadenas.

Veamos ahora la función `div` que permite realizar divisiones enteras:

```
*Main> div 45 12
3
```

Como vemos, recibe dos parámetros, el primero será el dividendo y el segundo el divisor.

```
*Main> 45 `div` 12
3
```

Aquí estamos haciendo exactamente la misma operación, pero con la función `div` aplicada de modo infijo. Para ello tenemos que poner el primer argumento, el nombre de la función entre commillas hacia la izquierda y por último el segundo argumento.

Veamos ahora la división con decimales, que se hace igual que en la mayoría de lenguajes, con `/`:

```
*Main> (/) 45 12
3.75
*Main> 45 / 12
3.75
```

Como se trata de un operador, para llamarlo de modo prefijo como en la primera línea del ejemplo, debemos ponerlo entre paréntesis `(/)`. Para usarlo de modo infijo, como vemos, no hacen falta las comillas, de hecho, si las ponemos, obtendremos un error.

```
*Main> 45 `/` 12

<interactive>:18:5: parse error on input ‘/’
```

Veamos algunos ejemplos más, y te invito a probar funciones y expresiones que veas por ahí:

```
*Main> 3 == 5
False
*Main> 3 == (6 `div` 2)
True
*Main> 3 == 3.0
True
```

La función `mod`, como en otros muchos lenguajes, calcula el signo de la división entera:

```
Prelude> mod 12 5
2
Prelude> 12 `mod` 5
2
```

La función `divMod` calcula simultáneamente el resultado de la división entera y el módulo, y los devuelve en forma de tupla (cociente, resto):

```
Prelude> divMod 12 5
(2,2)
Prelude>  12 `divMod` 3
(4,0)
Prelude>  12 `divMod` 2
(6,0)
Prelude>  13 `divMod` 2
(6,1)
```

Hablaremos de las tuplas más adelante, pues son un tipo heterogéneo, y nos permite almacenar datos con un cierto orden, pero que no tienen por qué ser del mismo tipo, por ejemplo, podemos guardar el nombre y la edad de diversas personas (a cual más rocosa en este caso):

```
Prelude> ("Chuck Norris", 75)
("Chuck Norris",75)
Prelude> ("Sylvester Stallone", 68)
("Sylvester Stallone",68)
```

La función `compare` recibe dos parámetros y nos devuelve cómo es el primero respecto al segundo; mayor (`GT`), igual (`EQ`), menor (`LT`):

```
*Main> compare 3 3
EQ
*Main> compare 3 3.0
EQ
*Main> compare 4 3
GT
*Main> compare 4 8
LT
```

Las funciones `succ` y `pred` devuelven, respectivamente, el sucesor y el predecesor de un número dado.

```
*Main> succ 78
79
*Main> pred 78
77
*Main> pred 0
-1
```

La función unaria `negate` cambia el signo a un número:

```
*Main> negate 13
-13
*Main> negate (-13)
13
*Main> negate -13

<interactive>:35:1:
    No instance for (Show (a0 -> a0)) arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
```

Como vemos, la función `negate` es simétrica ya que cumple la propiedad:

`negate x == negate (-x)` ó `negate (negate x) == x`

La función `abs` devuelve el valor absoluto de un número:

```
*Main> abs 554.71
554.71
*Main> abs (-554.71)
554.71
```

La función `signum` devuelve el signo de un número real expresado mediante un entero; -1 si el real es negativo, 0 si el real es 0, 1 si el real es positivo.

Las funciones `abs` y `signum` deben cumplir la siguiente ley:

```
abs x * signum x == x
```

`fromInteger` pasa de un entero al tipo que le especifiquemos:

```
*Main> fromInteger 1649725 :: Float
1649725.0
*Main> fromInteger 1649725 :: Rational
1649725 % 1
*Main> fromInteger 16 :: Rational
16 % 1
```

Nota: En los números racionales de Haskell (`Rational`), el `%` indica la raya de fracción.

```
*Main> sqrt (fromIntegral 16)
4.0
*Main> sqrt 16
4.0
```

En versiones de GHCi anteriores de 7.8.X la primera línea debía ser así obligatoriamente. Hoy en día hay una inferencia de tipos mejorada que nos permite programar más cómodamente, como en la segunda línea.

Comandos de GHCi:

- `Ctrl + L` es una combinación de teclas que **limpia cualquier consola en sistemas POSIX**, también funciona en GHCi. Es equivalente a `:! clear`.
- `:l nombre_fichero` sirve para **compilar** (e interpretar luego las aplicaciones que queramos) el fichero dado con nombre `nombre_fichero`.
- `:r` sirve para **recompilar** al vuelo el último fichero compilado.
- `:t expresión` nos permite **comprobar el tipo** de una expresión.
- `:k expresión_de_tipo` nos permite **comprobar el kind** de un tipo.
- `:i función` es una opción de GHCi que nos permite ver la **fijeza** (o asociatividad) de un operador.
- `:q` sirve para **salir** limpiamente de GHCi.

Definiendo nuestras primeras funciones

En la programación funcional, la principal actividad (y en realidad, lo único) que realizaremos será definir funciones. Para ello lo mejor es escribir primero una **declaración de tipos**:

```
ochenta :: Int
```

Las declaraciones de tipos suelen ser así:

```
nombre_funcion :: parametroDeTipo1 -> parametroDeTipo2 -> ... -> parametroDeTipo1N
```

Donde la función recibe un número N parámetros. De momento, vamos a pensar que el último parámetro es el tipo de retorno, por tanto nuestra función `ochenta` no recibe nada (no hay ninguna flecha) sino que devuelve un valor de tipo `Int`.Se puede leer como `ochenta` de tipo `Int`.

A continuación, escribimos la definición de `ochenta`:

```
ochenta = 80
```

Como habíamos dicho, devuelve un valor de tipo `Int`, en este caso un 80 programado duramente (sin calcularlo, simplemente escribiendo un inmediato). Veamos ahora la definición completa que nos permitirá ver la función con mayor claridad.

```
ochenta :: Int
ochenta = 80
```

Nota: es mejor ser "verbose" y poner las declaraciones de tipos de todas nuestras funciones, ya que nos ayudará para dos cosas; 1) es documentación implícita y 2) evita que el compilador infiera tipos más generales y no deseados debido a la falta de información de un código sin declaraciones de tipos.

Importante en Haskell, el signo `=` **no** significa asignación de variables, significa definir una **equivalencia**. Aquí estamos diciendo que la palabra `ochenta` es **equivalente** al literal **80**. Donde quiera que veas uno de los dos, lo puedes reemplazar por el otro y el programa siempre producirá la misma salida. Esta propiedad es conocida como **transparencia referencial** y es y será cierta para cualquier definición en Haskell, sin importar lo complicada que sea.

Definamos ahora una función `sumar` que reciba dos parámetros y los sume:

```
sumar :: Int -> Int -> Int
sumar a b = a + b
```

La función `sumar` la hemos implementado nosotros, pero Haskell ya contiene una función `add` que tiene el mismo efecto. Asimismo, podríamos usar la función `(+)` (y de hecho ya la estamos utilizando en `sumar`, que sólo es un wrapper).

```
*Main> sumar 4 5
9
*Main> (+) 4 7
11
*Main> 4 + 7
11
*Main> 4 `sumar` 11
15
```

Como vemos, en Haskell hay muchas maneras de llamar a las funciones, y de crear wrappers que nos harán la programación más cómoda y los nombres de las funciones fáciles de recordar.

Comprensiones de Listas

En la notación de conjuntos se usa una definición intensiva o por comprensión cuando se requiere que el lector (o el lenguaje de programación y por tanto, el ordenador) conozca las propiedades de los elementos de ese conjunto y los límites de generación.

En Haskell hay una herramienta muy poderosa que nos permite crear conjuntos que cumplan todas las restricciones que notros queramos, por ejemplo para resolver problemas con restricciones.

Por ejemplo, queremos saber cuáles son las longitudes de los lados de un subconjunto de los triángulos rectángulos cuya hipotenusa mida un máximo de 100 unidades métricas cualesquiera. Para ello escribimos una función:

```
comprension :: [(Int,Int,Int)]
comprension = [(a,b,c) | a <- [1..100], b <- [a + 1..100], c <- [b + 1..100], a^2 + b^2 == c^2]
```

En esta función nos damos cuenta de las siguientes cosas:

1. Su tipo retorno es una lista (entre corchetes) que contiene una tupla con tres `Int`. Estos tres `Int` son las longitudes de los tres lados de cada triángulo que entrará en la solución.
2. La comprensión se define entre corchetes, y está dividida en tres partes. 1) (antes de la barra vertical) Tipo que contendrá la lista solución, en este caso `(a,b,c)` 2) Elementos que se generarán. Están compuestos de un nombre al cual se van enlazando valores de la lista que le pasemos. Los generadores se separan por comas. `a <- [1..100], b <- [a..100], c <- [b..100]` 3) condición o condiciones, separadas también por comas, en este caso `a^2 + b^2 == c^2`.

Hemos usado un pequeño truco que es muy útil para estos casos y para la optimización de bucles en un lenguaje imperativo. Vemos que cuando b va a tomar un valor, su lista no empieza en `1`, sino en el actual valor de `a` incrementado en 1, y lo mismo ocurre para `c`, que empieza en el valor actual de `b` incrementado en 1. De este modo, no obtendremos repeticiones de triángulos en nuestra lista sin la necesidad de añadir condiciones adicionales.

Para tener una idea del orden en que se van creando las tuplas o listas que queramos, veamos un ejemplo:

```
*Main> [(a,b,c) | a <- [1,2], b <- [3,4], c <- [5,6]]
[(1,3,5),(1,3,6),(1,4,5),(1,4,6),(2,3,5),(2,3,6),(2,4,5),(2,4,6)]
```

Como vemos, primero se liga un valor al nombre `a`, después al `b` y por último, el que más cambia de una tupla a otra, será el `c`.

Las comprensiones de listas nos sirven para muchas cosas, por ejemplo, podemos hacer una función que comprueba si un número es primo. Además, esta función parará desde que encuentre un divisor en la lista de números desde `2` hasta `n-1`:

```
esPrimo :: Integer -> Bool
esPrimo n = null [k | k <- [2..n-1], n `mod` k == 0]
```

Aquí usamos otro pequeño truco, nos creamos una lista con todos los potenciales divisores del número, efectuamos la división, nos quedamos con el módulo y comprobamos si es cero (con lo cual sería divisible). Estos números irán a parar a la lista solución como `k`s, por tanto, desde que esa lista contenga un sólo elemento, ya el número no será primo.

La función `null` recibe una lista, devolviendo `True` si está vacía (por lo tanto `n` es primo) y `False` en caso de que contenga al menos un elemento.

Listas

Las listas son el tipo más importante para aprender programación funcional. Una lista de `Int` que contenga los valores 1,2,3 puede ser escrita como `1:2:3:[]`, o de una forma más azucarada sintácticamente, `[1,2,3]`. La principal propiedad de las listas es que son *homogéneas*, es decir, contienen ninguna, una o muchas (incluso infinitas) instancias *del mismo tipo*.

Por ello, puede haber listas de funciones, de enteros, de flotantes, de booleanos, y de todos los tipos que se nos puedan ocurrir, incluso de listas (formando listas de listas).

La lista vacía se representa con los corchetes sin nada en medio `[]`, y es el caso base típico de la recursividad en listas.

Nota importante: `[]` tiene dos significados en Haskell, puede ser o bien la lista vacía o bien el constructor de tipo lista. En otras palabras, el tipo `[a]` (lista de a) puede también ser escrito como `[] a`.

Las listas tienen muchas funciones útiles definidas exportadas por el módulo `Data.List`. Pasamos a ejemplificar algunas de ellas:

```
Prelude> head [1,2,3,4]
1
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> init [1,2,3,4]
[1,2,3]
Prelude> last [1,2,3,4]
4
```

Como vemos, `head` y `last` devuelven elementos, mientras que `init` y `tail` devuelven listas. Las cuatro funciones del ejemplo anterior generan una excepción si se ejecutan sobre la lista vacía.

elemIndex

La función `elemIndex` nos permite obtener el índice de la lista (recuerda que la cabeza tiene índice 0) que coincide con el valor del primer parámetro. El tipo de `elemIndex` es:

```
elemIndex :: Eq a => a -> [a] -> Maybe Int
```

Lo que significa que debemos usar la función sobre listas de tipos a los que se pueda aplicar la función (==) para comprobar si son iguales. Le pasamos un elemento de tipo `a` y una lista con elementos de tipo `a` (`[a]`) y nos devuelve la posición en la que se encuentra el elemento...si es que se encuentra en esa lista. Vemos un palabro extraño, ¿`Maybe`, quizá?, sí. `Maybe` es un constructor de tipos, no entraremos aún en ello, nos basta con saber que si `elemIndex` encuentra el elemento, devolverá `Just x`, donde x será el índice donde se encuentra el elemento. Si no lo encuentra, devolverá `Nothing`.

```
Prelude Data.List> elemIndex 12 [13,12,11,10]
Just 1
Prelude Data.List> elemIndex 'l' "Vamos a la playa"
Just 8
Prelude Data.List> elemIndex False [True,True,False,True]
Just 2
Prelude Data.List> elemIndex False [True,True,True,True]
Nothing
```

Reverse

`reverse` es una función que le da la vuelva a una lista, el primer elemento pasa a ser el último, el segundo el penúltimo, etc. No hay problema en ejecutarla sobre la lista vacía.

```
Prelude> reverse "Con los terroristas"
"satsirorret sol noC"
Prelude> reverse [1,2,3,4]
[4,3,2,1]
Prelude> reverse [True,False,True,False]
[False,True,False,True]
Prelude> reverse []
[]
```

Recursividad en listas:

En programación funcional, hablar de "iteraciones" es hablar de recursividad. Las listas se suelen procesar de manera recursiva, haciendo uso del reconocimiento de patrones, o de guardianes.

Una secuencia (que puede contener valores del tipo que queramos) es palíndroma si su primer carácter es igual al último, si el segundo es igual al penúltimo, etc. Todas estas igualdades se deben dar hasta que lleguemos a la cadena vacía, que es palíndroma. Si no se da alguna, la cadena no es palíndroma y no hará falta hacer más comprobaciones. Veamos cómo se hace esto en Haskell:

```
esPalindroma :: (Eq a) => [a] -> Bool
esPalindroma xs
  | null xs = True
  | head xs == last xs = esPalindroma (if not (null (init xs))
                                       then tail (init xs)
                                       else [])
  | otherwise = False
```

Vemos primero `(Eq a)`, que es una **clase de tipos**. Esta clase viene a imponer que el objeto tipo a, que, fijémonos, es el tipo de los elementos de la lista, debe pertenecer a la clase de tipos `Eq`. Esto viene a decir que para saber si una secuencia (lista) es palíndroma, debemos poder comprobar si sus elementos son iguales para afirmar igualdad o desigualdad.

Su tipo es `[a] -> Bool`, por tanto, dada una lista nos devolverá un valor de tipo `Bool` que indicará si es, o no, palíndroma.

La primera línea del cuerpo es el nombre de la función, `esPalindroma` separada por un espacio de xs. Luego, vemos que hay una cierta indentación (todas las `|` están a la misma altura), y esto son **guardianes**.

Los guardianes se ejecutan en orden, de arriba a abajo.

1. La función `null` devuelve `True` si la lista que le pasamos es vacía, y `False` si contiene algún elemento. Como habíamos dicho, la lista vacía es palíndroma, por tanto devolvemos `True` en caso de que xs sea vacía.
2. Empieza la recursividad: obtenemos la cabeza `head xs`, y el último `last xs` y comprobamos si son iguales; si son iguales: aplicamos recursivamente `esPalindroma`, haciendo un pequeño truco:

Los condicionales `if then else` (siempre debe haber un `else` en Haskell, no se puede omitir) devuelven la expresión que nosotros queramos, por tanto, como `init` y `tail` daba error al ser llamada sobre lista vacía, comprobamos si es vacía después de hacer `init xs`.

- Si **no** es vacía, llamamos a esPalindroma con `tail (init xs)`, es decir, le quitamos a la lista sus elementos primero y último, para seguir comprobando recursivamente.

- Si es vacía, devolvemos la lista vacía. Pero sabemos que si `init` devuelve `True` estamos ante una lista singleton, ¿por qué no podemos devolver directamente `False`? Recordemos que todo lo que hagamos dentro del `if then else` dará algo que se pasará como parámetro a `esPalindroma`, por lo cual estamos obligados a devolver una lista.

3. Si la lista no es vacía y cabeza y cola no son iguales, estamos ante una lista que no es palíndroma, y devolvemos `False`.

Una versión que usa la función `reverse`:

```
esPalindroma' :: (Eq a) => [a] -> Bool
esPalindroma' xs = xs == reverse xs
```

Ahora pasamos a ver dos versiones de la función que elimina duplicados de una lista dada:

Versión recursiva que sólo usa reconocimiento de patrones y el constructor `Cons`, también llamado `(:)`:

```
quitarDuplicados :: (Eq a) => [a] -> [a]
quitarDuplicados [] = []
quitarDuplicados [x] = [x]
quitarDuplicados (x:y:ys) = if x == y
                            then quitarDuplicados (y:ys)
                            else x : (quitarDuplicados (y:ys))
```

Una versión más corta, no recursiva, que hace uso de funciones predefinidas:

```
quitarDuplicados :: (Ord a) => [a] -> [a]
quitarDuplicados = map head . group . sort
```

Vagancia

Las personas tendemos a la procrastinación con facilidad. Como Haskell fue (y sigue siendo) hecho por personas, Haskell también es vago. A Haskell no le gusta trabajar por gusto, y esto permite que ciertas computaciones terminen sobre estructuras de datos teóricamente infinitas.

El ejemplo más típico de lista infinita es aquella generada por `[1..]`, que genera una lista infinita de números naturales en orden creciente empezando en el 1.

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21Interrupted.
```

Pero nosotros podemos parar la computación cuando lo necesitamos, por ejemplo mediante funciones que sólo necesiten cierta cantidad de elementos, como `take`:

```
Prelude> take 21 [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21]
```

A su vez, existen funciones condicionales que paran en cuanto cierta condición se cumple o se deja de cumplir, como `takeWhile` y `find`. El siguiente ejemplo añade elementos a la lista resultado hasta que encuentre algún número impar, en cuyo caso, termina:

```
Prelude> takeWhile even [1..]
[]
Prelude> takeWhile even ([2,4..20] ++ [21..])
[2,4,6,8,10,12,14,16,18,20]
```

Como vemos, podemos hacer de este modo un cierto filtrado de listas infinitas, o al menos, de su parte inicial.

La función `find` devuelve el primer elemento que cumple un predicado que le pasamos como primer parámetro, si éste existe, si no, devuelve `Nothing`. Notar que el tipo de `find` es `find :: (a -> Bool) -> [a] -> Maybe a`. `Maybe a` denota un tipo que podría haber fallado, como por ejemplo:

```
Prelude Data.List> find (==3) [0,2,4,6,8]
Nothing
```

En cambio, si encuentra el elemento lo envuelve en un contexto mínimo, a través del constructor `Just x`.

```
Prelude Data.List> find (>3) [0,2,4,6,8]
Just 4
```

Otra función útil para muchas cosas en `findIndex`, que es básicamente lo mismo que `find` pero devolviendo el índice donde se encuentra el elemento buscado (empezando por 0), o `Nothing` en el caso de que no lo encuentre:

```
Prelude Data.List> findIndex (=='x') "exoesqueleto"
Just 1
Prelude Data.List> findIndex (=='x') "servoarmadura"
Nothing
```

La siguiente función devuelve la posición de una letra en el alfabeto español, tomando la primera posición (‘A’) como posición 1:

```
posAlfabeto :: Char -> Int
posAlfabeto c = case findIndex (== (toUpper c)) alfabeto of Just x -> succ x
                                                            Nothing -> -1
  where
    alfabeto = ['A'..'M'] ++ ['Ñ'] ++ ['O'..'Z']

*Main> posAlfabeto '¿'
-1
*Main> posAlfabeto 'Ñ'
14
 map posAlfabeto "España"
[5,19,16,1,14,1]
```

cycle

La función `cycle` crea una lista infinita circular a partir de una lista dada:

```
*Main> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
```

Programación Origami: plegado/desplegado de listas:

Como muy bien ilustra esta fotografía del origami hecho por orudorumagi11 de deviantart.com, se pueden hacer cosas asombrosas simplemente plegando billetes de un dólar. Con las listas de Haskell también.

Nota: después de leer esta sección, recomiendo encarecidamente coger papel y bolígrafo para hacer trazas a mano de estas funciones.

Las funciones recursivas que hemos visto se pueden "resumir", es decir, mediante patrones recursivos varios, podemos definir funciones de manera muy directa.

Los patrones que podemos encapsular mediante plegados son aquellos que:

1. Ante lista vacía, devuelve el valor inicial del acumulador.
2. Ante listas no vacías, usamos el patrón (x:xs) para aplicar una función sobre un elemento (x) y llamar recursivamente a la misma función con el resto de la lista.

Las funciones de plegado permiten:

1. Reducir a un valor único una estructura de datos (como por ejemplo una lista)
2. Implementar rápido funciones que recorran una lista, elemento por elemento, y devuelvan un valor único (que puede ser una lista).

Esta familia de funciones necesita:

1. Una función binaria
2. Un valor inicial del acumulador
3. Una lista

Depende de dónde empiece el plegado (foldr va de derecha a izquierda y foldl va de izquierda a derecha) el primer o último elemento de la lista será evaluado junto con el acumulador para producir un valor. Este valor pasará a ser el nuevo acumulador y el desplegado continuará hasta que la lista termine, dando un valor único.

Desplegado generalizado de foldl:

Sea `f` una función y `z` el acumulador:

`foldl f z ['a', 'b', 'c']` se despliega de la siguiente manera:

```
\z f -> (f (f (f z a) b) c)
```

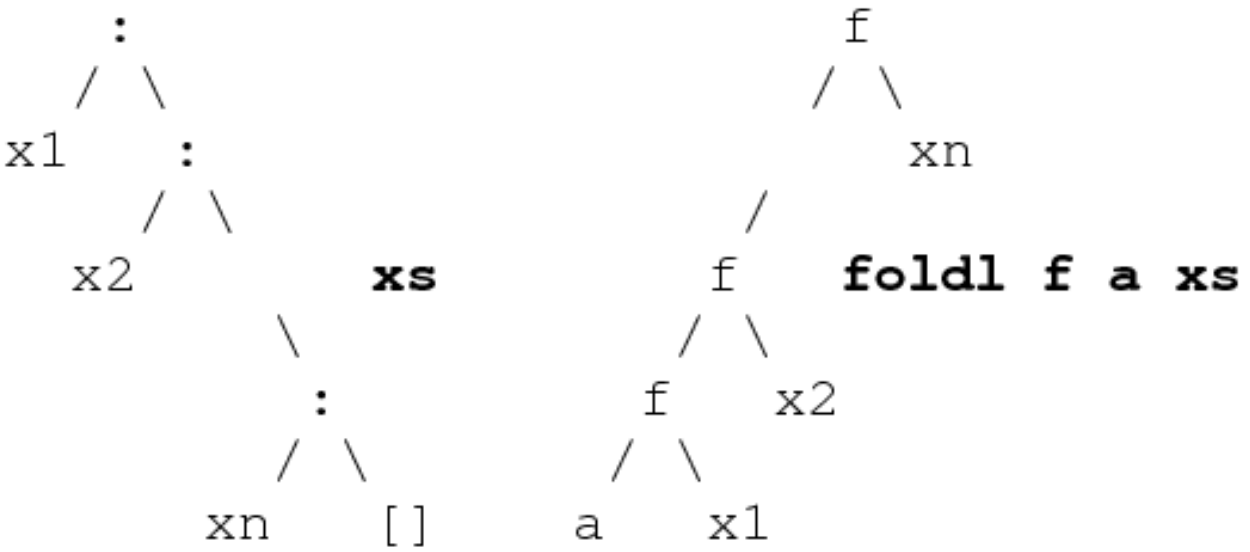

Figure 1: Despliegue de foldl

Un ejemplo de `foldl` en acción; la siguiente función pasa una lista (cuyos elementos deben ser enteros de una cifra) a un valor entero:

```
listaAInt :: [Int] -> Int
listaAInt = foldl (\z x -> 10*z + x) 0
```

Como vemos, es una implementación muy concisa. Podemos pensar en `foldl` como una especie de bucle for en programación imperativa, que va acumulando el resultado paso a paso, y ejecutando la misma operación hasta que llega al final de la lista.

Código de foldr

```
foldr k z = go
          where
            go []     = z
            go (y:ys) = y `k` go ys
```

Desplegado generalizado de foldr:

Sea `f` una función y `z` el acumulador:

`foldr f z ['a', 'b', 'c']` se despliega de la siguiente manera:

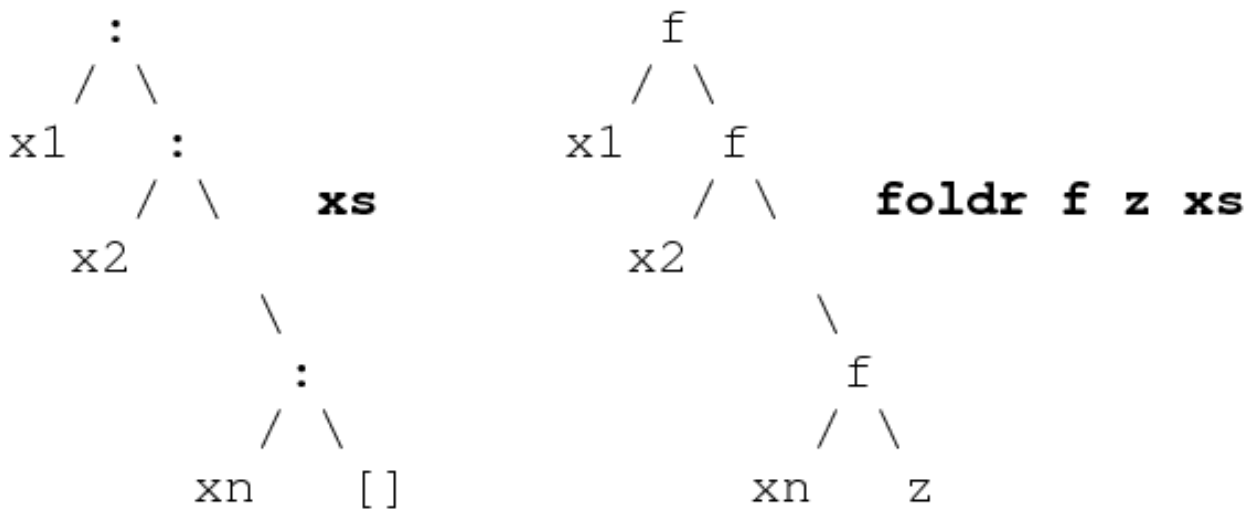

Figure 2: Despliegue de foldr

```
\f z -> (f a (f b (f c z)))
```

Como vemos, el `foldl` empieza por la izquierda de la lista mientras que `foldr` empieza por el final.

Truco: en el **foldr** (plegado a la **derecha**) el acumulador es el parámetro **derecho** (segundo parámetro de la función binaria), y en el **foldl** (plegado a la **izquierda**) el acumulador es el parámetro **izquierdo** (primer parámetro de la función binaria).

Estilo funcional: Generalmente, si tenemos una función del tipo `foo a = bar b a`, se podría reescribir como `foo = bar b` a causa de la "curryficicación".

Rendimiento: La función `(++)` (tiempo cuadrático) es mucho más lenta que `(:)` (tiempo lineal), así que normalmente se usa foldr cuando estamos construyendo una nueva lista a partir de otra.

Traza de foldl

Así evalúa Haskell la expresión `foldl (+) 0 [1,2,3]`:

```
foldl (+) 0 [1,2,3] =
foldl (+) (0 + 1) [2,3] =
foldl (+) ((0 + 1) + 2) [3] =
foldl (+) (((0 + 1) + 2) + 3) [] =
((0 + 1) + 2) + 3 =
(1 + 2) + 3 =
3 + 3 =
6
```

Como toda esta evaluación se hace de manera recursiva, `foldl` mantiene en memoria toda la construcción. Esto no es un problema cuando se trata de listas pequeñas, pero podría producir stack overflows si las listas son demasiado grandes.

Para resolver esto, tenemos la función `foldl'` definida en `Data.List`, que no mantiene en memoria las computaciones intermedias sino que las resuelve en el acumulador. Veámoslo con un ejemplo:

```
foldl' (+) 0 [1,2,3] =
foldl' (+) 1 [2,3] =
foldl' (+) 3 [3] =
foldl' (+) 6 [] =
6
```

También existe `foldl1'`.

Importante: `foldr` funciona para listas infinitas, mientras que `foldl` no.

foldr1 y foldl1:

Se suelen usar cuando la función no tiene sentido sobre listas vacías. La diferencia respecto a `foldr` y `foldl` es que no reciben acumulador, `foldr1` toma el último elemento (`last`) como acumulador inicial, mientras que `foldl1` toma el primer elemento (`head`).

Dan errores en tiempo de ejecución si se intentar aplicar a listas vacías, cosa que no ocurre con `foldr` y `foldl`.

Plegados desplegados (bolígrado y papel insaid):

Podemos ver los plegados como aplicaciones sucesivas de una función a los elementos de una lista.

Ejemplo 1: plegado por la derecha con una función binaria `f` y un acumulador inicial `z`.

Cuando aplicamos plegado por la derecha (`foldr`) sobre la lista `[3,4,5,6]` estamos realmente haciendo:

```
f 3 (f 4 (f 5 (f 6 z)))
```

`f` se llama primeramente con el último elemento de la lista (`last`, en el ejemplo "6") y el acumulador, entonces ese valor se le pasa como acumulador a la siguiente aplicación de `f`, hasta que la lista termine.

Si `f` = `(+)` y `z` = 0 entonces:

```
3 + (4 + (5 + (6 + 0)))
```

Si escribimos `(+)` como una función prefija:

```
(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))
```

De modo similar, hacer un plegado por la izquierda (`foldl`) sobre la lista `[3,4,5,6]` con `g` como función binaria y `z` como acumulador es equivalente a hacer:

```
g (g (g (g z 3) 4) 5) 6
```

Si usamos `flip (:)` como función binaria y `[]` como acumulador, eso es equivalente a lo siguiente:

```
flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6
```

recuerda: `x:[] = [x]`.

Desarrollamos paso a paso y obtenemos:

```
flip (:) (flip (:) (flip (:) 3:[] 4) 5) 6

flip (:) (flip (:) 4:[3] 5) 6

flip (:) 5:[4,3] 6

[6,5,4,3]
```

Definimos la función `and'` que recibe una lista de `Bool` y devuelve `True` si todos son `True`, y `False` en cualquier otro caso.

```
and' :: [Bool] -\> Bool
and' = foldr (&&) True
```

Si desarrollamos paso a paso sobre la lista `[True,False,True]` obtenemos:

```
True && (False && (True && True))
```

El último `True` es nuestro acumulador inicial, el resto viene de la lista de entrada. Si evaluamos esta expresión, dará `False`.

¿Qué ocurre si aplicamos esta función sobre una lista infinita, como puede ser `repeat False`, que tiene infinitos elementos, todos ellos `False`?. Obtendríamos algo como:

```
False && (False && (False && False ...
```

Haskell es vago (lazy), así que solo computará lo realmente necesario. Veamos cómo funciona la función `&&`:

```
(&&) :: Bool -\> Bool -\> Bool
True && x = x
False && _ = False
```

Como vemos, desde que ve que el primer argumento es `False`, no necesita mirar el otro, simplemente devuelve `False`.

Por tanto en nuestro ejemplo con valores sin fin en el que el primero es `False`, el segundo patrón se cumple, retornando `False` sin que Haskell necesite evaluar el resto de la lista infinita.

Por tanto, `foldr` funcionará sobre listas infinitas cuando la función binaria que le pasemos no necesite evaluar siempre su segundo parámetro para dar un resultado fijo. Por ejemplo, a `&&` no le importa el valor de su segundo parámetro si el valor del primer parámetro es `False`.

Unfolds

Scans

Las funciones `scanl` y `scanr` son como `foldl` y `foldr`, excepto que devuelven todos los estados intermedios del acumulador en forma de lista.

Scanl deja el resultado final en el último elemento de la lista resultante.

Scanr lo deja en la cabeza de la lista.

Pueden ser utilizados para monitorizar el progreso de funciones definidas mediante plegados.

Operador de aplicación de funciones ($)

```
($) :: (a -\> b) -\> a -\> b
f $ x = f x
```

No es simplemente aplicar funciones sobre un parámetro. La aplicación normal de funciones (mediante un espacio) tiene una precedencia muy alta, mientras que la aplicación mediante `$` tiene la menor precedencia.

El operador predefinido `$` se comporta como la función unaVez y por tanto, permite evitar paréntesis gracias a su baja prioridad. Importante: el primer argumento de `$` debe ser una función y se puede llamar de diversas maneras:

```
($) inc 5, (inc $) 5, ($ 5) inc
```

Y se cumplen las siguientes igualdades:

```
($) f = (f $) = f
($ x) = flip ($) x = \\f -\> f x
```

La aplicación de funciones mediante un espacio es asociativa a izquierdas:

```
f a b c es lo mismo que ((f a) b) c)
```

La aplicación de funciones mediante el dólar `($)` es asociativa a derechas:

```
f $ g $ x es lo mismo que f $ (g $ x)
```

Por tanto, podemos reescribir `sum (filter (\> 10) (map (*2) [2..10]))` como:

```
sum $ filter (\> 10) $ map (*2) [2..10]
```

También podemos usar la aplicación parcial de `$` para, por ejemplo, mapear la aplicación de funciones sobre una lista de funciones.

```
ghci\> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Si recuerdas el cuerpo de la función `$`, deducirás que está fijando el segundo parámetro (aplicación parcial) y sólo queda la función, que se va cogiendo una por una de la lista, y aplicándose.

Tipos creados por el usuario:

Los constructores de valor son funciones como cualquier otra. Reciben ciertos parámetros y devuelven un valor de un tipo. En los módulos que creemos podemos exportarlos o no, depende de lo que queramos. Si no exportamos estos constructores, nuestra clase será más abstracta y podremos cambiar su implementación cuando queramos, siempre quemantengamos el comportamiento y las declaración de tipos de las funciones intactos, pues podrían haber sido usados en código anterior por los usuarios de nuestro módulo.

Los constructores de tipos tienen que tener rellenos todos los parámetros de tipo.

No existe un tipo `Maybe`, sino `Maybe Int`, `Maybe Char`…etc.

Las listas también usan parámetros de tipo, no existe el tipo `[]` sino `[Int]`, `[Char]`, `[[(String,Int)]]`…etc.

Decimos que un tipo es concreto si no recibe parámetros de tipo (como `Char` o `Bool`), o si recibe parámetros de tipo pero todos están ya rellenados. Para cualquier valor, su tipo es siempre un tipo concreto.

Lo mejor es no poner restricciones en las declaraciones de datos, incluso si parece tener sentido. Necesitarás ponerlas en las declaraciones de tipos de funciones aunque hayas puesto las restricciones en la declaración de datos.

Clases de tipos

Las clases de tipos son un tipo de interfaz que define ciertos comportamientos, y un tipo puede hacerse instancia de una clase de tipos si soporta ese comportamiento.

Por ejemplo el tipo `Int` es instancia de la clase de tipos `Eq` porque la clase de tipos `Eq` define el comportamiento para las cosas de las cuales se puede comprobar si son iguales o distintas. Como es posible comprobar si dos enteros son iguales, `Int` forma parte de la clase de tipos `Eq`.

La utilidad real de todo esto es que las funciones `==` y `/=` vienen definidas en la clase de tipos `Eq` y por tanto podemos usarla en los `Int` y el resto de tipos dentro de la clase de tipos Eq. Debido a esto, expresiones como `"freinn" == "guapo"` (`True` de toda la vida), ó `8 == 7` son aceptadas.

Por tanto, las clases de tipos no tienen demasiado que ver con las clases de los lenguajes imperativos orientados a objetos (C++, Ruby…).

Clase Ord

En una declaración como esta:

```
data Bool = False | True deriving (Ord)
```

¿Qué constructor de valor da una instancia menor? La respuesta es sencilla, el que está definido primero, más a la izquierda (en este caso `False`).

```
ghci> True `compare` False
GT

ghci> True > False
True

ghci> True < False
False
```

Esto es así porque ninguno de estos constructores de valor tiene parámetros de tipo, en caso de que los tuvieran, se compararían estos parámetros (para ello deben ser derivados de la clase de tipos `Ord`).

En el tipo de dato `Maybe a`, el constructor de valores `Nothing` se especifica antes del constructor de valores `Just valor`, así que el valor de `Nothing` es siempre más pequeño que el valor de `Just algo`, incluso si ese algo es menos un billón de trillones. Pero si especificamos dos `Just valores`, entonces comparará lo que hay dentro de ellos.

```
ghci> Nothing < Just 100
True

ghci> Nothing > Just (-49999)
False

ghci> Just 3 `compare` Just 2
GT

ghci> Just 100 > Just 50
True
```

Nota comparar funciones no es posible, pues no son instancias de `Ord`.

Sinónimos de tipo

La palabra reservada **type** quizás no esté del todo bien elegida, pues los sinónimos no crean ningún tipo nuevo.

Lo que realmente hacen es llamar a un tipo existente de otra manera, lo cual debería servir para dar un significado más claro a los tipos para un potencial lector de nuestro código.

El ejemplo más claro se ve en la definición de `String` en Haskell:

```
type String = [Char]

toUpperString :: [Char] -> [Char]
```

Puede ser escrita de forma más legible y clara:

```
toUpperString :: String -> String
```

Los sinónimos se usan para dar información adicional del contenido de nuestros datos.

Por ejemplo:

```
ListaTelefonos :: [(String, String)]
ListaTelefonos =
  [("betty", "555-2938")
  ,("bonnie", "452-2928")
  ,("patsy", "493-2928")
  ,("lucille", "205-2928")
  ,("wendy", "939-8282")
  ,("penny", "853-2492")
  ]
```

Puede ser convertido a:

```
type NumeroTelefono = String
type Nombre = String
type ListaTelefonos = [(Nombre, NumeroTelefono)]
```

A continuación crearemos una función que devolverá si una combinación de `Nombre` y `NumeroTelefono` pertenece a nuestra `ListaTelefonos`.

```
inListaTelefonos :: Nombre -> NumeroTelefono -> ListaTelefonos -> Bool
inListaTelefonos nombre numerotel listatel = (nombre, pnumber) `elem` listatel
```

Parametrizando sinónimos de tipo

Los sinónimos pueden ser parametrizados. Si queremos un tipo que representa una lista de asociación pero queremos que sea general y use cualquier tipo de claves y de valores, podemos hacer:

```
type TablaHash k v = [(k, v)]
```

Podríamos definir un mapa de `Int` a lo que sea que queramos:

```
type IntMap v = Map Int v
```

Cuando vayamos a implementar esto, probablemente queramos hacer un `qualified import de Data.Map`. Cuando lo hacemos, los constructores de tipo también necesitan estar precedidos por un nombre de módulo.

Como podemos aplicar parcialmente funciones para obtener nuevas funciones, podemos hacer lo mismo para los constructores de tipos, obteniendo nuevos constructores de tipos parcialmente aplicados:

```
type IntTablaHash = Map.Map Int
```

Es importante entender bien la diferencia entre los constructores de tipos y los de valor.

Sólo con sinónimos **no** es posible hacer `TablaHash [(4,6),(5,7),(8,9)]`.

Lo único que podemos hacer es referirnos a su tipo usando nombres diferentes. Por ejemplo `[(4,6),(5,7),(8,9)] :: TablaHash Int Int`, cuyo efecto es que los números de los pares tengan tipo `Int`.

Por tanto, los sinónimos de tipo y los tipos en general pueden ser usados en la porción de tipos de Haskell. Es decir:

- declaraciones `type` y `data`
- después del `::` en declaraciones de tipo o anotaciones de tipo

Constructores de tipos

Constructores de valor

Tipo de dato Either

`Either` en inglés puede significar:

- conjunción: o
- pronombre: uno u otro cualquiera de los dos
- adverbio: también

En Haskell se define como sigue:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Tiene dos constructores de valor. Si se usa `Left`, su contenido será de tipo `a`. Por el contrario, si se usa `Right`, sus contenidos serán del tipo `b`.

Por tanto haremos reconocimiento de patrones en ambos `Left` y `Right`, haciendo diferentes cosas depende de cuál sea reconocido.

Se utiliza porque el tipo `Maybe a` sólo nos proporciona información de que ha habido un fallo en la ejecución de la función (ni siquiera sabemos a priori qué fallo). `Maybe a` es útil para funciones que sólo pueden fallar de una manera (por ejemplo, un `find` que no encuentra lo que busca en el dato que le pasamos). Normalmente se usan para funciones que sólo pueden fallar de una manera, o cuando no estamos interesados en el fallo que se produjo.

Si tenemos interés en *cómo y por qué* la función falló, entonces deberíamos usar `Either a b` tomando:

- a: tipo que informa del fallo
- b: tipo de la computación exitosa

Tipos recursivos

Piensa en esta lista: `[5]`. Es simplemente azúcar sintáctico para `5:[]`. En el lado izquierdo de `:`, hay un valor, en el lado derecho, hay una lista. En este caso es una lista vacía. ¿Qué pasa con la lista `[4,5]`? Bien, si le quitamos el azúcar se convierte en `4:(5:[])`. Mirando el primer `:`, vemos que también tiene un elemento a su izquierda y una lista, `(5:[])`, a su derecha. Lo mismo ocurre con algo como `3:(4:(5:6:[]))`, que podría reescribirse como `3:4:5:6:[]` (porque `:` es asociativo a la derecha) ó `[3,4,5,6]`.

Por tanto se deduce que una lista puede ser:

- una lista vacía
- un elemento unido mediante `:` a otra lista (que puede ser la lista vacía)

Implementemos nuestra lista mediante tipos algebraicos:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Usemos record syntax para esclarecer un poco qué ocurre aquí:

```
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord)
```

Podría confundirte el constructor `Cons`. `Cons` es otra forma de decir `:`. En listas, `:` es un constructor que recibe un valor y una lista y devuelve otra lista. En otras palabras, tiene dos campos:

- a
- List a

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

Hemos llamado a nuestro constructor `Cons` de manera infija de modo que se puede ver que equivale a `:`. `Empty` es como `[]`, y `4 Cons (5 Cons Empty)` equivale a `4:(5:[])`.

Constructores de datos infijos

ImportanteCualquier función (o constructor) que tenga en su nombre únicamente carácteres especiales será infijo automáticamente.

aquí definimos un nuevo operador, `:-:`:

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

- infixr n op: significa asociativo a la derecha.
- infixl n op: significa asociativo a la izquierda.
- la n establece la prioridad, es decir, cuanto mayor sea, antes se debe hacer su operación.

La fijeza del operador `*` es `infixl 7 *`, y la del operador `+` es `infixl 6`. Esto significa que ambos son asociativos a la izquierda (por tanto, `4 * 3 * 2` es equivalente a `(4 * 3) * 2`, pero une los operandos de manera más estricta que `+`, porque tiene una mayor fijeza. Así que `5 * 4 + 3` es equivalente a `(5 * 4) + 3`.

Importante: El *reconocimiento de patrones* en realidad es el reconocimiento de *constructores*.

Maybe

`Maybe Int` es un tipo concreto, pero `Maybe` es un constructor de tipo que recibe un tipo como parámetro.

Entrada salida:

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

- `IO tipo_retorno`: acción de entrada/salida junto al tipo que produce como resultado.
- `()`: tupla vacía, también llamada *unidad*, es un tipo y un valor a la vez.

Por tanto, la función `putStrLn` recibe una String y devuelve una acción de entrada/salida que *produce* un resultado de tipo `()`.

Una acción de entrada/salida es algo que cuando se lleva a cabo, lleva a cabo una acción con un efecto lateral y además presentará un resultado.

Decimos que una acción de E/S *produce* este resultado. Como imprimir una cadena a la terminal no devuelve ningún tipo de valor con sentido, se devuelve una unidad `()`.

```
main = do
    putStrLn "Hola, escribe tu nombre:"
    nombre <- getLine
    putStrLn ("Me caes bien, " ++ nombre ++ "!")
```

Ahora vemos que nuestro programa a compilar usa una nueva sintaxis, la sintaxis `do`. En ella, se especifican las acciones a realizar línea por línea.

La sintaxis do lo que hace es "pegar" las acciones, una detrás de otra, para convertirlas en una única acción de E/S. **El tipo de la acción resultado coincide con el tipo de la última acción especificada**.

Por tanto, `main` siempre tiene una declaración de tipos de la forma `main :: IO something` donde `something` es algún tipo concreto.

```
ghci> :t getLine
getLine :: IO String
```

Podemos leer la sentencia `name <- getLine` como: haz la acción de E/S `getLine`, y después liga su valor resultado a `name`. Como `getLine` tiene un tipo de `IO String`, name tendrá un tipo `String`.

El constructo `<-` sirve para "sacar" los datos que ha conseguido nuestra acción y ligarlos a algún lado. Sólo podemos "sacar" datos de una acción E/S cuando estamos "dentro" de otra acción E/S.

Hemos mencionado el concepto de *pureza*, pero, ¿qué es realmente? Se trata de una propiedad de las funciones, en concreto que devuelvan lo mismo cada vez que son llamadas, cosa que no pasa con las funciones que trabajan con la E/S.

Volvamos a la sentencia `name <- getLine`. ¿Qué tipo tendrá `name`? La respuesta es que se trata de una `String` como otra cualquiera. De acuerdo a lo que hemos estudiado, ¿Es la siguiente sentencia válida?

```
nameTag = "Hello, my name is " ++ getLine
```

La respuesta es **no** por las siguientes razones:

1. (++) necesita dos listas del mismo tipo. En esta sentencia, el primer parámetro es de tipo `String`, mientras que el segundo es el tipo `IO String`. Recuerda que no se pueden concatenar Strings y acciones E/S.
2. Siempre que queramos "sacar" los datos de una acción E/S, debemos usar el constructo `<-`.

```
main = do
  foo <- putStrLn "Hola, escribe tu nombre:"
  name <- getLine
  putStrLn ("Me caes bien, " ++ nombre ++ "!")
```

En este caso hemos ligado a `foo` el resultado producido por `putStrLn "Hello, what's your name?"` mediante el constructo `<-`. ¿Tiene esto sentido? Bien poco, puesto que dicho resultado es la tupla vacía o unidad `()`.

La última acción no se liga a ningún nombre, debido a que el propio bloque `do` se encarga de extraer su resultado y produce ese resultado como propio.

Excepto la última línea, podemos ligar a un nombre todas las demás si así lo deseamos o es más conveniente.

Lo único que hace la siguiente línea es darle otro nombre a la acción de E/S:

```
myLine = getLine
```

Por tanto, las únicas cuatro maneras que tenemos de hacer acciones de E/S son:

1. Ligar una acción E/S al nombre `main`.

2. Estar dentro de una acción mayor que ha sido compuesta mediante un bloque `do`.
3. Se ejecutan los bloques que "caen" dentro de `main`, y un bloque puede contener otros definidos previamente.
4. Ejecutándolas en una línea de GHCi. El propio GHCi aplica `show` al valor resultante, y después lo imprime en la terminal mediante la acción `putStrLn`.

Let dentro de acciones E/S

`Let` se usa para enlazar *valores puros* a nombres dentro de bloques `do`. Por tanto, `let` se usa para ligar expresiones puras. **No** se trata de una acción E/S.

Por tanto:

- Cuando queramos ligar resultados de una acción E/S a un nombre, usaremos el constructo `<-`.
- Cuando queramos ligar a un nombre valores puros, usaremos `let`.

Uso de recursividad y de return

```
main = do
  linea <- getLine
  if null linea
    then return ()
    else do
      putStrLn $ invertirPalabras linea
      main

invertirPalabras :: String -> String
invertirPalabras = unwords . map reverse . words
```

Respecto al `else`: unimos acciones E/S en un bloque do porque después del else sólo puede haber una acción E/S.

La recursividad es válida aquí ya que es una acción E/S aplicando otra acción E/S.

La palabra reservada `return` podría confundirnos bastante, sobre ella, debemos decir:

- Se trata de transformar un valor puro, en este caso la tupla vacía o unidad, `()`, en una acción que al ejecutarse *"no hace nada"*, sólo devolver un valor tipo `()` en este caso. Si hubiéramos hecho `return "hola"`, hubiera devuelto un valor de tipo `IO String`, sin ejecutar realmente ninguna acción E/S.
- Es necesario su uso en el ejemplo que hemos puesto para lidiar con la cadena vacía. Debemos ejecutar una acción, pero sobre la cadena vacía no vamos a hacer nada, por ello, devolvemos una acción que no hace nada, salvo devolver un resultado del tipo que más nos convenga.
- **No** provoca que el bloque `do` en el que se encuentra termine su ejecución.

Este programa no termina su ejecución hasta llegar a la última línea:

```
main = do
  return ()
  return "JAJAJA"
```

```
  line <- getLine
  return "BLA BLA BLA"
  return 4
  putStrLn line
```

Por tanto, lo que hace `return` es crear acciones que devuelven un resultado, el cual se pierde al no estar ligado a un nombre. Es por esto que podemos decir que es justo lo contrario que el constructo `<-`.

Resumen de lo expuesto:

- Constructo `<-`: *"desempaqueta"* un valor de una acción y lo liga a un nombre.
- `return`: *"empaqueta"* el resultado de una acción.

Por tanto, si primero empaquetamos con `return` y después desempaquetamos con `<-`, estamos haciendo algo equivalente a los `let` dentro de bloques `do`:

```
main = do
  a <- return "hell"
  b <- return "yeah!"
  putStrLn $ a ++ " " ++ b
```

Es por tanto equivalente a:

```
main = do
  let a = "hell"
  b = "yeah"
  putStrLn $ a ++ " " ++ b
```

Por tanto, `return` puede ser usada para:

- Crear acciones que no hagan nada salvo devolver un resultado del tipo que le pasemos.
- Permitir que un bloque `do` devuelva el resultado que nosotros queramos, poniendo el return que devuelve un valor del tipo deseado al final del bloque `do`.

Funciones útiles para la E/S

`putStr`

Recibe un valor tipo `String` o `[Char]` y **devuelve una acción** que imprime por pantalla (consola) la cadena argumento.

`putStr`

```
main = do
  putStr "Hola, "
  putStr "¡buenos "
  putStrLn "días!"
```

Tiene un efecto parecido a `putStrLn`, pero `putStr` no empieza una nueva línea tras imprimir por consola la cadena argumento.

Recuerda: devuelve una acción.

Su salida sería:

```
Hola, ¡buenos días!
```

`putChar`

```
main = do
  putChar 't'
  putChar 'e'
  putChar 'h'
```

Recibe un carácter y **devuelve una acción** que imprime dicho carácter por pantalla (a través de la consola).

Podemos definir `putStr` como la aplicación recursiva de `putChar`:

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
  putChar x
  putStr xs
```

De este modo vemos que podemos usar definiciones recursivas en funciones que implican E/S.

`print`

La mejor manera de definirla es usando composición `putStrLn . show`.

Aquí vemos que como lo último que se ejecuta es `putStrLn`, se devuelve también una acción.

GHCi está todo el tiempo usando `print` cada vez que le pasamos un valor concreto:

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["ola","k","ase"]
["ola!","k!","ase!"]
ghci> print $ map (++"!") ["ola","k","ase"]
["ola!","k!","ase!"]
```

`when` (parte de Control.Monad)

```
import Control.Monad

main = do
  input <- getLine
  when (input == "freinn") $ do
    putStrLn input
```

El programa anterior es equivalente a:

```
main = do
  input <- getLine
  if (input == "SWORDFISH")
    then putStrLn input
    else return ()
```

Por tanto `when` es útil cuando queremos que cierta acción E/S se ejecute en ciertos casos, y en los casos contrarios no hacer nada.

`sequence`

Recibe una lista de acciones E/S y devuelve una acción E/S que hará todas esas acciones una detrás de otra. El tipo de retorno será una lista con todos los resultados que produjo cada acción.

`map print [1,2,3,4]` no creará una acción E/S, sino una lista de acciones E/S.

La sentencia anterior es equivalente a:

```
[print 1, print 2, print 3, print 4]
```

Si queremos transformar una lista de acciones E/S en una acción E/S, debemos secuenciarlas con `sequence`:

```
ghci> sequence $ map print [1,2,3,4,5]
1
2
3
4
5
[(),(),(),(),()]
```

En GHCi, se muestran todos los resultados de las acciones…¿todos? te preguntarás, la respuesta es no. Se muestran todos los resultados distintos de (), la archiconocida unidad o tupla vacía. Debido a esto se imprimió en el ejemplo anterior `[(),(),(),(),()]`, por ser distinto resultado de `()`.

- Imprimir "hola", imprimirá hola y nada más, puesto que devuelve `()`, y `()` no se imprime.
- La función `getLine`, se imprime su resultado, pues es una IO String, distinta de `()`.

`mapM`

Mapear una función que devuelve una acción E/S sobre una lista y después secuenciar es muy común, debido a ello, `mapM` y `mapM_` fueron creadas.

`mapM` hace lo anteriormente mencionado, y `mapM_` hace lo mismo pero sin devolver nunca el resultado, por tanto, se usa `mapM_` cuando no nos importa el resultado de la secuencia de acciones.

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
```

En caso de que no nos importe el resultado de la secuencia de acciones:

```
ghci> mapM_ print [1,2,3]
1
2
3
```

forever

La función `forever` recibe una acción de E/S y devuelve una acción E/S que simplemente repite la acción E/S recibida eternamente. Se encuentra en `Control.Monad`.

El siguiente programa pedirá entrada a un usuario y la devolverá pasada a mayúsculas.

```
import Control.Monad
import Data.Char

main = forever $ do
  putStr "Introduce una cadena de carácteres: "
  l <- getLine
  putStrLn $ map toUpper l
```

forM

Es básicamente lo mismo que `mapM` pero con el orden de los parámetros cambiados. Por ello, recibe una lista y la función a mapear, que será luego secuenciada.

```
import Control.Monad

main = do
  colores <- forM [1,2,3,4] (\a -> do
    putStrLn $ "¿Qué color asocias con el número "
             ++ show a ++ "?"
    color <- getLine
    return color)
  putStrLn "Los colores que asociaste a 1, 2, 3 y 4 son: "
  mapM putStrLn colores
```

La lambda `(\a -> do ...)` recibe un número y devuelve una acción E/S (cuyo resultado es la String del color que hayamos elegido). La última acción del bloque `do` va a definir el tipo del bloque entero. Por ello, se ha hecho un `return` color.

Sin embargo, en este ejemplo esto no era necesario, puesto que `color <- getLine` simplemente "saca" el resultado de `getLine` y lo liga a un nombre (en este caso color). Recordemos que el `return` va a devolver la acción que no hace nada y devuelve como resultado el tipo del parámetro que le pasemos. Por tanto el programa anterior puede ser escrito de forma más cómoda:

```
import Control.Monad

main = do
  colores <- forM [1,2,3,4] (\a -> do
    putStrLn $ "¿Qué color asocias con el número "
```

```
            ++ show a ++ "?"
    getLine)
  putStrLn "Los colores que asociaste a 1, 2, 3 y 4 son: "
  mapM putStrLn colores
```

Como `getLine`:

- devuelve una acción que lee una línea de consola.
- produce un resultado con la cadena leída.

Hemos podido ahorrarnos una línea y hacer nuestro programa más compacto y legible.

La función `forM` (llamada con sus dos parámetros) produce una acción E/S, cuyo resultado ligamos a colores. color es simplemente una lista normal que contiene valores de tipo `String`. Al final, imprimimos todos esos colores, uno en cada línea, llamando a `mapM putStrLn colores`.

Puedes pensar en `forM` como decir, "Crea una acción E/S para cada elemento de esta lista". Lo que hará cada acción E/S puede depender de el elemento usado para crear la acción. Finalmente, ejecuta esas acciones y liga su resultado a algo". (Aunque no es necesario ligar esos resultados si no queremos).

Podríamos haber conseguido el mismo resultado sin `forM`, pero es más legible con `forM`.

Uso típico: normalmente, se usa `forM` cuando queremos mapear y secuenciar ciertas acciones que definimos "al vuelo" usando notación `do`.

Ideas importantes sobre la E/S en Haskell

Las acciones de E/S son valores casi como cualquier otro. Las podemos pasar como parámetros a funciones, y las funciones pueden devolver acciones E/S como resultado/s. Lo que tienen de especial es que si "caen" dentro de la función `main` o se ejecutan en GHCi, se llevan a cabo. Ahí es cuando se nota su acción (nunca mejor dicho).

Cada acción E/S puede también producir un resultado para contarte qué ha obtenido del "mundo real".

Ficheros y flujos

Un flujo es una sucesión de trozos de datos que van entrando o existen en un programa a lo largo del tiempo.

Cuando introducimos carácteres de entrada a través del teclado, podemos pensar en esos carácteres como un flujo.

Cree un fichero de texto llamado `amayusculas.hs` con el siguiente contenido:

```
import Control.Monad
import Data.Char

main = forever $ do
  l <- getLine
  putStrLn $ map toUpper l
```

A continuación, compílelo de la siguiente forma:

```
$ ghc --make amayusculas.hs
```

Redirección de la entrada

Ahora usaremos el operador < (como regla nmemotécnica, véalo como la punta de una flecha, que envía flujo de un lado al otro) para redirigir el flujo del teclado al fichero que queramos:

```
$ ./amayusculas < ejemplo.txt
```

El resultado es que imprimirá por pantalla (consola) el contenido del fichero fuente, pasado a mayúsculas.

La función `getContents` permite trabajar con flujos. Su comportamiento consiste en leer un flujo hasta que encuentre el carácter fin de fichero (EOF, normalmente Ctrl + D).

El programa anterior, ahora implementado con `getContents`:

```
import Data.Char

main = do
  contents <- getContents
  putStr $ map toUpper contents
```

De este modo, la entrada del teclado se redirige al fichero ejemplo.txt, de modo que el programa leerá el fichero, y no el teclado, hasta que encuentre el carácter de fin de fichero (Ctrl + D).

```
$ ./amayusculas < ejemplo.txt
```

Cree un fichero con el siguiente contenido y llámelo `lineas.txt`

```
soy corta
y yo
soy una línea laaaaaaaarga!!!
sí, soy larga, y qué jajajajaja!!!!!!
línea corta
laaaaaaaaaaaaaaaaaaaaaaaaarga
corta
```

A continuación, cree el siguiente programa, cuyo fichero de código he llamado `sololineascortas.hs`

```
main = do
  contents <- getContents
  putStr (soloLineasCortas contents)

soloLineasCortas :: String -> String
soloLineasCortas = unlines . filter (\line -> length line < 10) . lines
```

Compílelo de la siguiente manera:

```
$ ghc --make sololineascortas.hs
```

Y por último pásele el contenido del fichero como entrada:

```
$ ./sololineascortas < lineas.txt
```

Como el patrón de obtener un contenido, procesarlo mediante una función de tipo `(String -> String)` y luego imprimir el resultado de la función es tan común, hay una función que se encarga de realizar esta secuencia de operaciones.

Dicha función se llama `interact` y recibe una función de tipo `(String -> String)`. El programa anterior se podría acortar haciendo uso de `interact`:

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly = unlines . filter (\line -> length line < 10) . lines
```

El siguiente programa pide una `String`, comprueba si es palíndroma o no e imprime el resultado:

```
main = interact informarPalindromas

informarPalindromas :: String -> String
informarPalindromas =
  unlines .
  map (\xs -> if esPal xs then "palíndroma" else "no palíndroma") .
  lines

esPal :: String -> Bool
esPal xs = xs == reverse xs
```

Cuando funciona por fichero, pasa las líneas a una lista, mapea una lambda que comprueba si es palíndroma e imprime el resultado, y luego lo pasa todo a líneas de nuevo.

```
ghc --make palindromas.hs
```

Cree un fichero de ejemplo, con el siguiente contenido:

```
alcorque
radar
rayar
casa
rotor
salas
```

Ejecute el programa redirigiendo la entrada al fichero `palabras.txt`:

```
./palindromas < palabras.txt
```

Que produce la siguiente salida:

```
no palíndroma
palíndroma
palíndroma
no palíndroma
palíndroma
palíndroma
```

Cree un fichero llamado `agujero_negro.txt` con el siguiente contenido:

"Un agujero negro u hoyo negro es una región finita del espacio en cuyo interior existe una concentración de masa lo suficientemente elevada como para generar un campo gravitatorio tal que ninguna partícula material, ni siquiera la luz, puede escapar de ella."

Y un programa que nos permita mostrar su contenido por consola:

```
import System.IO

main = do
  handle <- openFile "agujero_negro.txt" ReadMode
  contenidos <- hGetContents handle
  putStr contenidos
  hClose handle
```

Analicemos el comportamiento de la función `openFile` empezando por su cabecera:

```
openFile :: FilePath -> IOMode -> IO Handle
```

Por tanto, recibe un `FilePath` (ruta del fichero) y un `IOMode` (modo de E/S), lo abre y produce un resultado de tipo `IO Handle`.

`Filepath` es simplemente un sinónimo para `String` cuyo cometido es dar información sobre lo que se espera recibir.

```
type FilePath = String
```

En cambio, `IOMode` es un tipo, que funciona con los 4 constructores de valor siguientes:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Remarcar que es simplemente una enumeración, al contrario de lo que sería `IO Mode`, que sería si existiera, una acción que produce un valor de tipo Mode.

Finalmente, el resultado de `openFile` será una acción de E/S que abrirá el archivo especificado en el modo especificado y luego devolverá un valor de tipo `Handle`. Si ligamos ese resultado de la acción a algo, obtendremos un `Handle`, el cual representa dónde está nuestro fichero. Usaremos ese handle para saber de qué fichero leer.

Un handle es un indicador de la posición de fichero que debemos leer, no es contenido. El contenido está en el fichero y el handle indica por dónde vamos a empezar a leer.

En la siguiente línea, tenemos una función llamada `hGetContents`. Recibe un `Handle`, así que sabe de qué fichero obtener los contenidos, y devuelve una `IO String`, es decir, una acción que devuelve el contenido del fichero leído como una String.

Nota: la diferencia respecto a `getContents` es que `hGetContents` espera un `Handle` para leer desde un fichero, mientras que `getContents` lee automáticamente desde la entrada estándar. Salvo eso, funcionan igual.

Lo mejor de `getContents` y `hGetContents` es que no leen todo el fichero cargándolo en memoria, sino que sólo lo leen cuando realmente es necesario. Por ello, podemos tratar `contenidos` como el *contenido completo del fichero*, sin ocupar demasiada memoria, perfecto para lidiar con ficheros grandes.

Con `putStr contenidos`, imprimimos el contenido por la salida estándar, y después hacemos `hClose`, la cual recibe un handle y devuelve una acción E/S que cierra el fichero.

Es necesario cerrar **siempre** los ficheros que abramos con `openFile`, puesto que si no lo hacemos nuestro programa podría terminar al intentar abrir un fichero cuyo handle no ha sido cerrado.

Función withFile

Su declaración de tipos es la siguiente:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
```

Recibe una ruta a un archivo, un `IOMode`, y una función que recibe un handle y devuelve cierta acción.

Su comportamiento es devolver una acción que:

1. Abra el fichero.
2. Haga algo con el fichero
3. Cierre el fichero

Si algo va mal, `withFile` se asegura de que el handle del archivo se cierre.

```
import System.IO

main = do
    withFile "girlfriend.txt" ReadMode (\handle -> do
    contents <- hGetContents handle
    putStr contents)
```

`(\handle -> ...)` es la función que recibe un handle y devuelve una acción, se suele hacer con una lambda. Necesita recibir una función que devuelva una acción de E/S, en vez de sólo recibir una acción E/ S, hacerla y luego cerrar el fichero, porque la acción E/S que le pasáramos no sabría sobre qué fichero operar.

De esta forma, `withFile` abre el fichero y le pasa el handle a la función que le pasemos. Recibe una acción E/S de esa función y después hace una acción E/S que es como la acción original, pero también se asegura de que el handle quede cerrado.

Función bracket

Se trata de una función definida en el módulo `Control.Exception`. Tiene la siguiente declaración de tipos:

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

Su primer parámetro es una acción E/S que adquiere un recurso, como un handle de fichero. Su segundo parámetro es una función que libera ese recurso. Estas función es llamada incluso si ocurre una excepción. El tercer parámetro es donde ocurre lo principal, como leer de un fichero o escribir en él.

Como `bracket` va de adquirir un recurso, hacer algo con él, y asegurar que será liberado, implementar `withFile` es realmente sencillo:

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile name mode f = bracket (openFile name mode)
    (\handle -> hClose handle)
    (\handle -> f handle)
```

El primer parámetro que le pasamos a `bracket` abre el archivo, y da como resultado un handle de archivo. El segundo parámetro recibe ese handle y lo cierra. `bracket` asegura que esto ocurre incluso si ocurre una excepción. Finalmente, el tercer parámetro recibe un handle y le aplica la función `f` a él, la cual recibe un handle de fichero y hace cosas con ese handle, como leer o escribir en el fichero correspondiente.

Más funciones que trabajan con handles

Se trata de funciones como `hGetLine`, `hPutStr`, `hPutStrLn`, `hGetChar`.

- `hGetLine`: recibe un handle y devuelve una acción E/S que lee una línea de ese fichero.
- `hPutStrLn`: recibe un handle y una `String` y devuelve una acción que escribirá esa `String` en el fichero asociado al handle y añade un carácter de nueva línea ('\n') después de dicha `String`.

Todas se comportan igual que sus casi homónimas (excepto por la 'h' inicial), salvo que todas ellas reciben un handle y actúan sobre el fichero asociado a ese handle.

Funciones que cargan ficheros y tratan su contenido como cadenas

Se trata de las funciones `readFile`, `writeFile` y `appendFile`.

readFile

La declaración de tipos de la función `readFile` es:

```
readFile :: FilePath -> IO String
```

`readFile` es una función que lee el fichero de la ruta especificada en su argumento `FilePath` (recuerda que es sólo un sinónimo de `String`) y devuelve una acción de E/S que leerá ese fichero (de manera vaga) y ligará su contenido a algo como una `String`.

El ejemplo anterior con `readFile`:

```
import System.IO

main = do
  contents <- readFile "girlfriend.txt"
  putStr contents
```

Debido a que no obtenemos un handle con el cual identificar nuestro fichero, no podemos cerrarlo manualmente, así que Haskell hace eso por nosotros cuando usamos `readFile`.

writeFile

La función `writeFile` tiene una declaración de tipos así:

```
writeFile :: FilePath -> String -> IO ()
```

Recibe la ruta a un fichero y una cadena a escribir en dicho fichero, y devuelve una acción E/S que hará la escritura. Si ese fichero ya existe, será reiniciado antes de escribirse de nuevo. Este código lee un fichero, pasa su contenido a mayúsculas y lo escribe en un nuevo fichero:

```
import System.IO
import Data.Char

main = do
  contents <- readFile "agujero_negro.txt"
  writeFile "agujero_negromayus.txt" (map toUpper contents)
```

appendFile

La función `appendFile` tiene la misma declaración de tipos que `writeFile` y actúa casi igual. La diferencia está en que `appendFile` no borra el fichero si éste ya existía, sino que empieza a escribir a partir del carácter fin de fichero.

El siguiente programa leerá la línea que escribamos en consola y la añadirá a nuestra lista de tareas en `tareas.txt`:

```
import System.IO
main = do
  cosaQueHacer <- getLine
  appendFile "tareas.txt" (cosaQueHacer ++ "\n")
```

Como `getLine` no nos da el carácter '\n' incluido en la cadena resultado, lo añadimos nosotros.

```
import System.IO
import System.Directory
import Data.List

main = do
  contenido <- readFile "tareas.txt"
  let cosasQueHacer = lines contenido
    cosasNumeradas = zipWith (\n linea -> show n ++ " - " ++ linea)
                             [0..] cosasQueHacer
  putStrLn "Estas son tus cosas que hacer:"
  mapM_ putStrLn cosasNumeradas
  putStrLn "¿Cuál quieres borrar?"
  cadenaNumero <- getLine
  let number = read cadenaNumero
    nuevasCosasQueHacer = unlines $ delete (cosasQueHacer !! number) cosasQueHacer
  (tempName, tempHandle) <- openTempFile "." "temp"
  hPutStr tempHandle nuevasCosasQueHacer
  hClose tempHandle
  removeFile "tareas.txt"
  renameFile tempName "tareas.txt"
```

1. Leemos `tareas.txt` y lo ligamos a `contenido`.
2. Partimos el contenido en una lista de strings, con una línea por string. Por tanto `cosasQueHacer` es ahora algo como esto:

```
["Fregar la loza", "Limpiar al perro", "Sacar la ensalada del horno"]
```

3. Aplicamos `zipWith` para asociar un número a cada tarea en `cosasQueHacer` creando `cosasNumeradas`.
4. Usamos `mapM_ putStrLn cosasNumeradas` y `putStrLn "¿Cuál quieres borrar?"` Para informar al usuario de lo que debe hacer.
5. Usamos `cadenaNumero <- getLine` y `let number = read cadenaNumero` para quedarnos con el número elegido por el usuario.
6. Usamos la función `openTempFile`. Esta función recibe:

- La ruta a un directorio temporal.

- Un nombre de plantilla para un fichero.

Y abre un fichero temporal. Usamos `.` como directorio porque en muchos sistemas denota el directorio actual. La plantilla "temp" indica que el fichero temporal que se creará tendrá por nombre "temp" seguido de algunos carácteres aleatorios. Devuelve una acción E/S que crea el fichero temporal, y cuyo resultado será un par (tupla con dos elementos), que contiene: el nombre del fichero temporal y un handle.

7. Ahora que tenemos un fichero temporal abierto, escribimos `nuevasCosasQueHacer` en dicho fichero temporal. De modo que el fichero original permanece sin cambios, y el fichero temporal contiene las líneas del original excepto la que hemos borrado.
8. Cerramos los dos ficheros y borramos el original con `removeFile`, la cual recibe la ruta de un fichero y lo borra.
9. Después de borrar el antiguo `tareas.txt`, usamos `renameFile` para renombrar el fichero temporal como `tareas.txt`.

Nota: `removeFile` y `renameFile` reciben rutas de fichero, no handles, como parámetros.

Importante: el motivo del uso de `openTempFile` suele ser evitar sobreescribir o ensuciar ficheros importantes.

Todo funciona, pero la vida es dura, y por ello, si algo va mal después de abrir nuestro fichero temporal, el programa termina, pero el fichero temporal no se limpia. Arreglemos eso.

Limpiando

Usaremos `bracketOnError` del módulo `Control.Exception`. Es muy similar a `bracket`, pero `bracket` adquirirá un recurso y después se asegurará de hacer siempre limpieza después de usarlo, `bracketOnError` hará la limpieza con sólo producirse una excepción.

```
import System.IO
import System.Directory
import Data.List
import Control.Exception

main = do
  contents <- readFile "tareas.txt"
  let cosasQueHacer = lines contents
      cosasNumeradas = zipWith (\n linea -> show n ++ " - " ++ linea)
                               [0..] cosasQueHacer
  putStrLn "Estas son tus cosas que hacer:"
  mapM_ putStrLn cosasNumeradas
  putStrLn "¿Cuál quieres borrar?"
  cadenaNumero <- getLine
  let numero = read cadenaNumero
      nuevasCosasQueHacer = unlines $ delete (cosasQueHacer !! numero) cosasQueHacer
  bracketOnError (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
      hClose tempHandle
      removeFile tempName)
    (\(tempName, tempHandle) -> do
      hPutStr tempHandle nuevasCosasQueHacer
      hClose tempHandle
      removeFile "tareas.txt"
      renameFile tempName "tareas.txt")
```

En vez de usar `openTempFile` normalmente, la usamos con `bracketOnError`. Después, escribimos lo que queremos que pase si hay un error; en este caso, queremos cerrar el handle del fichero temporal y luego borrar el fichero temporal. Finalmente, escribimos lo que queremos hacer con el fichero temporal cuando todo va bien, y esas líneas son las mismas que en el ejemplo anterior. Escribimos las nuevas cosas que hacer, cerramos el handle del fichero temporal, borramos nuestro fichero actual y renombramos el fichero temporal con el nombre del original.

Argumentos de línea de comandos

A la hora de automatizar tareas, es mucho más sencillo pasar argumentos a los programas por línea de comandos que tener que ejecutar el programa, que el programa pida los datos, etc.

El módulo `System.Environment` tiene dos acciones de E/S que son útiles para obtener argumentos de la línea de comandos: `getArgs` y `getProgName`.

getArgs

Su declaración de tipos es la siguiente:

```
getArgs :: IO [String]
```

Por tanto es una acción E/S que obtiene los argumentos con los que el programa se ejecutó y producirá una lista con dichos argumentos.

getProgName

Su declaración de tipos es:

```
getProgName :: IO String
```

Como vemos, devuelve una acción que obtiene el nombre del programa y produce una `String` con valor el nombre del programa.

Veamos con un ejemplo cómo funcionan las dos funciones anteriores:

```
import System.Environment
import Data.List

main = do
  args <- getArgs
  progName <- getProgName
  putStrLn "Los argumentos son:"
  mapM putStrLn args
  putStrLn "El nombre del programa es:"
  putStrLn progName
```

Usando la línea de comandos para algo útil

La línea de comandos nos permite introducir opciones, es decir, que un mismo programa pueda hacer varias cosas (normalmente tiene un propósito general y se divide en funciones más concretas).

Ejemplos de ejecución del programa que haremos:

Añadir la acción que queramos al fichero que queramos:

```
$ ./todo add todo.txt "Aprender Haskell bien aprendido"
```

Ver las tareas, comando view:

```
$ ./todo view todo.txt
```

Para eliminar una tarea, usamos su índice (recuerda que las listas empiezan en la posición 0):

```
$ ./todo remove todo.txt 2
```

Definamos la función dispatch, que recibirá una `String` y una lista de `String` y devolverá la acción de E/S adecuada ejecutándose sobre la lista de `String` que fue su segundo argumento. Esta acción producirá como resultado la tupla vacía unidad `()`. Remarcar que esta función está "curryficada". Lo mejor que tiene es que es muy sencillo añadirle funcionalidades al programa. Bastaría con definir una función y un "comando" en la función dispatch que la ejecute.

```
dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove
```

La función `main` queda definida de manera bastante simple, es sólo obtener la cabeza de los argumentos (la operación a realizar) y la cola de la lista de argumentos, y con estos dos valores llamar a `dispatch`.

```
main = do
  (command:argList) <- getArgs
  dispatch command argList
```

La función `add`:

```
add :: [String] -> IO ()
add [nombreFichero, cosaQueHacer] =
  appendFile nombreFichero("\n" ++ cosaQueHacer ++ "\r")
```

El programa completo quedaría así:

```
import System.Environment
import System.Directory
import System.IO
import Data.List
import Control.Exception
```

```
dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove

main = do
  (command:argList) <- getArgs
  dispatch command argList

add :: [String] -> IO ()
add [nombreFichero, cosaQueHacer] =
  appendFile nombreFichero ("\n" ++ cosaQueHacer ++ "\r")

view :: [String] -> IO ()
view [nombreFichero] = do
  contenido <- readFile nombreFichero
  let cosasQueHacer = lines contenido
      cosasNumeradas = zipWith (\n linea -> show n ++ " - " ++ linea)
                               [0..] cosasQueHacer
  putStr $ unlines cosasNumeradas

remove :: [String] -> IO ()
remove [nombreFichero, numeroString] = do
  contenido <- readFile nombreFichero
  let cosasQueHacer = lines contenido
      cosasNumeradas = zipWith (\n linea -> show n ++ " - " ++ linea)
                               [0..] cosasQueHacer
  let numero = read numeroString
      newTodoItems = unlines $
        delete (cosasQueHacer !! numero) cosasQueHacer
  bracketOnError (openTempFile "." "temp")
    (\(tempName, tempHandle) -> do
      hClose tempHandle
      removeFile tempName)
    (\(tempName, tempHandle) -> do
      hPutStr tempHandle newTodoItems
      hClose tempHandle
      removeFile nombreFichero
      renameFile tempName nombreFichero)
```

Lidiar con entrada errónea

Los usuarios no siempre ejecutan los programas de manera correcta, pues errar es humano. Sin embargo, si queremos un software que realmente sea robusto, debemos tener esto en cuenta y remediarlo de manera elegante.

Para conseguir esto, añadimos *"catchballs"*, es decir; patrones que "capturen" entrada errónea y actúen en consecuencia.

Por ejemplo, modifiquemos la función `dispatch` para que muestre un error en el caso de que no reconozca el comando que se le pasa:

```
dispatch :: String -> [String] -> IO ()
```

```
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove
dispatch comando = noExiste comando

noExiste :: String -> [String] -> IO ()
noExiste comando _ = putStrLn $ "El comando " ++
                     comando ++ " no existe"
```

También podemos añadir un "catchball" para el caso en el cual la función `add` sea llamada con una lista de parámetros con largo distinto de dos:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
add _ = putStrLn "The add command takes exactly two arguments"
```

Aleatoriedad

Es de sobra conocido que los ordenadores actuales unidos a nuestro conocimiento actual de las matemáticas y la informática, no permiten la generación de números realmente aleatorios, sino pseudoaleatorios.

Se necesita entrada externa (como la hora del sistema) para conseguir un grado de aleatoriedad si bien no total, suficiente.

La transparencia referencial es una de las características distintivas de Haskell, recordemos, este concepto hace referencia al hecho de que llamar a una función (en realidad, aplicar una función) con los mismos parámetros dos o más veces deberá dar siempre el mismo resultado.

La transparencia referencial nos permite:

- Razonar sobre los programas.
- Diferir la evaluación hasta que realmente sea necesaria.

Como hemos comentado, necesitamos E/S para traer aleatoriedad desde fuera a a nuestro programa.

El módulo `System.Random` tiene todas las funciones dedicadas a generar valores aleatorios.

random

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

Como vemos, hay dos nuevas clases de tipos implicadas:

- `RandomGen`: tipos que pueden actuar como una fuente de aleatoriedad.
- `Random`: tipos cuyos valores pueden ser aleatorios (booleanos, números,...etc.)

Centrémonos en lo que recibe y devuelve:

```
g -> (a, g)
```

Recibe un generador de valores aleatorios y devuelve un par compuesto por un valor aleatorio y un nuevo generador.

mkStdGen

```
mkStdGen :: Int -> StdGen
```

Sirve para producir un generador manualmente. Recibe un `Int`, y en base a él nos devuelve un generador.

Aunque hoy en día no es necesario indicarle a Haskell qué tipo queremos por resultado, es mejor hacerlo para que se ajuste más a nuestros propósitos:

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-3633736515773289454,693699796 2103410263)
```

Recuerda: la cota máxima y mínima de los tipos de la clase de tipos `Bounded` se puede obtener mediante `minBound` y `maxBound`. Ejemplo para Int: `minBound :: Int`.

¿Qué ocurre si volvemos a ejecutar el mismo comando?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-3633736515773289454,693699796 2103410263)
```

Como vemos, con un mismo generador se obtiene el mismo resultado. Probemos con un generador distinto.

```
ghci> random (mkStdGen 123456) :: (Int, StdGen)
(7397425676655451845,1471560279 2103410263)
```

Ahora usaremos una *anotación de tipo* diferente para obtener diferentes tipos de esa función:

```
ghci> random (mkStdGen 123456) :: (Float, StdGen)
(0.5922903,76514111 1655838864)
ghci> random (mkStdGen 123456) :: (Integer, StdGen)
(7397425676655451845,1471560279 2103410263)
ghci> random (mkStdGen 123456) :: (Bool, StdGen)
(True,645041272 40692)
```

Creemos un programa que nos permita tirar 3 monedas y ver los resultados:

recuerda: el `let` va indentado respecto a la función, y sus valores definidos todos deben estar en la misma columna coincidiendo con la inicial.

```
import System.Random

tresMonedas :: StdGen -> (Bool, Bool, Bool)
tresMonedas gen =
  let (primeraMoneda, nuevoGen) = random gen
      (segundaMoneda, nuevoGen') = random nuevoGen
      (terceraMoneda, nuevoGen'') = random nuevoGen'
  in  (primeraMoneda, segundaMoneda, terceraMoneda)
```

Importante: no hemos tenido que llamar `random gen :: (Bool, StdGen)` ya que hemos puesto una declaración de tipos que indica que queremos tres Booleanos, y así Haskell puede inferir que queremos valores booleanos.

Aleatorios en un rango

```
randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)
```

Como se deduce de su declaración de tipos, le pasamos un par con valor mínimo y máximo respectivamente, y nos devuelve un valor aleatorio en ese rango.

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

También está `randomRs`, que nos da una lista infinita de aleatorios en el rango que nosotros queramos.

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

```
import System.Random
main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
```

Hasta ahora hemos creado nuestros números aleatorios mediante enteros fijos en nuestros programas. Esto no es deseable, pues produce siempre los mismos resultados.

Para remediarlo, usaremos la función `getStdGen`, la cual lo que hace es pedir al sistema datos iniciales y usarlos para iniciar el generador global. Cuando ligamos `getStdGen` a un nombre, dicha función le pide al generador global un valor. Su declaración de tipo deja claro que devuelve una acción E/S que produce un valor de tipo `StdGen`.

```
getStdGen :: IO StdGen
```

```
import System.Random
main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
```

Probemos el programa:

```
$ ./random_string
pybphhzzhuepknbykxhe
$ ./random_string
eiqgcxykivpudlsvvjpg
$ ./random_string
nzdceoconysdgcyqjruo
$ ./random_string
bakzhnnuzrkgvesqplrx
```

Pero debemos tener cuidado, si hacemos dos llamadas a `getStdGen`, la función pedirá el mismo generador global dos veces, generando los mismos resultados.

```
import System.Random
main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
  gen2 <- getStdGen
  putStr $ take 20 (randomRs ('a','z') gen2)
```

La mejor forma de obtener dos cadenas diferentes es usar la acción `newStdGen`, la cual parte el generador aleatorio actual en dos generadores. Actualiza el generador aleatorio global con uno de ellos y produce el otro como resultado.

```
import System.Random
main = do
  gen <- getStdGen
  putStrLn $ take 20 (randomRs ('a','z') gen)
  gen' <- newStdGen
  putStr $ take 20 (randomRs ('a','z') gen')
```

No solo obtenemos un nuevo generador aleatorio cuando ligamos `newStdGen` a algo, sino que el el generador global también se actualiza. Esto significa que si llamamos de nuevo a `getStdGen` y lo ligamos a algo, obtendremos un generador que no es el mismo que `gen`.

Función reads

La función `reads` es muy buena para evitar errores en la entrada por teclado. Si el usuario introduce una entrada inválida, la función devuelve una lista vacía. En caso contrario, devuelve un par que contiene: la entrada correcta y el resto de la cadena introducida (por si queremos seguirla procesando).

```
import System.Random
import Control.Monad(when)

main = do
  gen <- getStdGen
  preguntarNumero gen

preguntarNumero :: StdGen -> IO ()
preguntarNumero gen = do
  let (numAleatorio, nuevoGen) = randomR (1,10) gen :: (Int, StdGen)
  putStrLn "¿En qué número del 1 al 10 estoy pensando? "
  cadenaNumero <- getLine
  when (not $ null cadenaNumero) $ do
    case reads cadenaNumero :: [(Integer,String)] of
      [(n, _)] -> if numAleatorio == fromIntegral n
                  then putStrLn "¡Correcto!"
                  else putStrLn $ "Lo siento, era " ++ show numAleatorio
      _        -> putStrLn "invalid input"
    preguntarNumero nuevoGen
```

Bytestrings

Procesar listas como cadenas tiene una desventaja: tiende a ser lento. Las listas son realmente vagas. Recuerda que una lista como [1,2,3,4] es azúcar sintáctico para 1:2:3:4:[]. El primer elemento de la lista

es forzosamente evaluado, pero el resto de la lista es simplemente una promesa de lista. En inglés, esas promesas son llamadas *thunk*.

Una *thunk* es una computación diferida. Haskell consigue ser vago usando *thunks* y computándolas sólo cuando debe, en vez de computarlo todo. Así que puedes pensar en las listas como promesas de que el siguiente elemento será enviado desde que deba ser así, y junto con él, la promesa del siguiente elemento después de él.

Como se puede suponer, no es una técnica demasiado eficiente, Por tanto, *para leer archivos* es mejor usar `bytestrings`. Las `bytestrings` son como listas, sólo que cada elemento en ellas tiene el tamaño de un btye (8 bits). Además, el modo en que son vagas también es distinto.

Falta la parte de bytestring del capítulo 9

Funtores

Los funtores son cosas que pueden ser mapeadas, como listas, `Maybes` y árboles. En Haskell, son descritos como la clase de tipos `Functor`, la cual sólo tiene un método de clase de tipos: `fmap`. `fmap` tiene una declaración de tipos así:

```
fmap :: (a -> b) -> f a -> f b
```

La cual se puede enunciar en el lenguaje natural de la siguiente manera: "Dame una función de a hasta b y una caja con un `a` (o varios) y te daré una caja con un `b` (o varios)". Es decir, aplica la función dada a los elementos que estén dentro de la caja.

Los valores de los funtores se pueden ver como valores con un *contexto añadido*. Por ejemplo, los valores `Maybe` tienen el contexto añadido de poder haber fallado. Las listas tienen el contexto añadido de que pueden contener muchos valores a la vez o ninguno. `fmap` aplica una función a esos valores preservando su *contexto añadido*.

Nota: los valores de funtor se pueden ver como "algo que se puede mapear".

Las acciones de E/S `IO` también pueden ser mapeadas con `fmap`.

Los funtores equivalen a la composición, son casi lo mismo:

Cabecera de `fmap`:

```
fmap :: (a -> b) -> f a -> f b
```

Sustituímos f por (->) r:

```
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
```

Pasamos a infija:

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Como vemos, ha quedado una función casi igual que la composición, pero con otras letras.

```
instance Functor ((->) r) where
    fmap = (.)

ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (+100) 1
"303"
```

En el tercer diálogo vemos claramente que al aplicarlo de forma infija se parece muchísimo a la composición.

Como todos los funtores, se puede pensar en las funciones como valores con contexto. Cuando tenemos una función como `(+3)`, podemos ver el valor como el eventual resultado de la función, y el contexto como que necesitamos aplicar la función a algo para obtener el resultado.

Todas las funciones en Haskell realmente reciben un único parámetro. Por tanto una función `a -> b -> c` recibe un único parámetro de tipo a y devuelve una función `b -> c`, que coge un parámetro y devuelve `c`. Por tanto, la aplicación parcial de funciones devuelve una función que toma los parámetros que dejamos sin "rellenar". Así que `a -> b -> c` puede ser reescrita como `a -> (b -> c)`.

Si le aplicamos esto a `fmap` obtenemos:

```
fmap :: (a -> b) -> (f a -> f b)
```

Por tanto es una función que recibe una función de `a` hasta `b` y devuelve otra función que recibiendo un valor de funtor sobre a devuelve un valor de funtor sobre `b`.

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

Puedes pensar en `fmap` de dos maneras, siendo las dos correctas:

1. Como una función que recibe una función y un valor de funtor y luego mapea esa función sobre el valor de funtor.
2. Como una función que recibe una función y la "eleva" para operar sobre valores de funtor.

El tipo `fmap (replicate 4) :: (Functor f) => f a -> f [a]` significa que la función funcionará con cualquier funtor. Lo que haga dependerá del funtor. Si usamos `fmap (replicate 3)` en una lista, se escogerá la implementación de `fmap` para listas, la cual es simplemente `map`. Si la usamos sobre `Maybe a`, aplicará `replicate 3` a el valor dentro de `Just`. Si es `Nothing`, seguirá siéndolo.

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) (Right "blah")
```

```
Right ["blah","blah","blah"]
ghci> fmap (replicate 3) Nothing
Nothing
ghci> fmap (replicate 3) (Left "foo")
Left "foo"
```

Leyes de los funtores

`fmap` se debe comportar respecto a los funtores de la siguiente manera. Si yo aplico `fmap` a un funtor, simplemente es como aplicar map sobre el funtor, nada más. Los funtores de la librería estándar obedecen dos leyes, y cuando creemos un funtor deberíamos verificar que las cumple para que sea instancia de funtor.

Recuerda: la función `id` es la *función identidad*, que simplemente devuelve su parámetro sin modificar. Si la queremos escribir como lambda, sería (\x -> x).

Ley 1

Si mapeamos la función `id` sobre un funtor, el valor del funtor que nos devuelve debería ser igual al valor original del funtor.

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

Veamos la implementación de `fmap` para `Maybe`:

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

Por tanto, se puede deducir que `fmap id` va a devolver:

En su primera ecuación: fmap id `(Just x)` dará `Just (id x)`, que puede reescribirse como (Just x). En su segunda ecuación: fmap id `Nothing` dará `Nothing`.

Por tanto se ve claramente que `fmap id = id`.

Ley 2

`fmap (f . g) = fmap f . fmap g`.

O, escrito de otra forma, para cualquier valor de funtor x, lo siguiente se debe cumplir:

`fmap (f . g) x = fmap f (fmap g x)`.

Es decir, un funtor es una cosa que puede ser mapeada. Si lo mapeamos, nos devuelve otra cosa que también puede ser mapeada (esto es, un funtor).

Esto permite encadenar tantas composiciones como queramos.

Volvamos a la demostración de esta propiedad con `Maybe`:

`fmap (f . g) Nothing` = `Nothing`.

`fmap f (fmap g Nothing)` = `Nothing`.

Ahora un poco más difícil, con un valor `Just`:

`fmap (f . g) (Just x)` viendo la implementación de `fmap` para `Maybe`:

`Just ((f . g) x)` que es `Just (f (g x))`.

Si usamos `fmap f (fmap g (Just x))` vemos en la implementación de `fmap` para `Maybe`:

`fmap g (Just x)` es `Just (g x)` lo que sustituyendo en la primera da:

`fmap f (Just (g x))` que aplicando la implementación queda:

`Just (f (g x))`.

Hay que comprobar si nuestro tipo que intenta ser un funtor realmente cumple las dos leyes de los funtores, y buscar contraejemplos de ello, si se encontraran, quiere decir que usar estos tipos como si de funtores se tratara podría resultar en código con fallos.

Funtores aplicativos

¿Qué pasa si mapeamos una función que recibe dos parámetros sobre un funtor?

Si hacemos `fmap (*) (Just 3)`, ¿Qué obtenemos? Si miramos la implementación de instancia de `Maybe` para `Functor`, sabemos que si es un valor `Just`, aplicará la función sobre el valor dentro del `Just`. Por tanto, hacer fmap (*) (Just 3) resulta en `Just` ((*) 3)), que también puede ser escrito como `Just` (3 *) si usamos secciones. Por tanto tenemos una función envuelta en un `Just`.

Veamos algunos ejemplos:

```
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])
ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) => [a -> a -> a]
```

Si mapeamos `compare`, que tiene tipo `(Ord a) => a -> a -> Ordering`, sobre una lista de carácteres, obtenemos una lista de funciones Char -> Ordering, porque la función `compare` se aplica parcialmente con los carácteres de la lista. No es una lista del tipo `(Ord a) => a -> Ordering`, puesto que la primera `a` aplicada fue un `Char`, así que la segunda debe ser también de tipo `Char`.

Por tanto, vemos que si mapeamos funciones "multiparámetro" sobre valores funtor, obtenemos valores funtor que contienen funciones.

Una ventaja de esto es que podemos mapear funciones que reciban esas funciones como parámetros, ya que lo que haya dentro del valor funtor será pasado a la función que estamos mapeando sobre él, como parámetro.

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

Pero, ¿qué ocurre si tenemos un valor de funtor de `Just (3 *)` y un valor de funtor de `Just 5`? Con funtores normales sólo podemos mapear funciones normales sobre funtores existentes. Incluso cuando mapeamos antes `\f -> f 9` sobre un funtor, estábamos mapeando una función normal. No podemos mapear una función que esté dentro de un valor de funtor sobre otro valor de funtor con lo que nos ofrece `fmap`.

Applicative

Lo primero que tenemos que tener claro es qué es `Applicative`; se trata de una clase de tipos para funtores.

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

La primera línea es la definición de la clase `Applicative`, y también introduce una restricción de clase. La restricción dice que si queremos hacer un constructor de tipo parte de la clase de tipos `Applicative` debe estar primero en `Functor`.

Por ello, sabemos que si algo es `Applicative` también es `Functor`, con lo cual podemos aplicar `fmap` sobre él.

El primer método que define es llamado `pure`. Su declaración de tipos es `pure :: a -> f a`, `f` sería nuestra instancia de funtor aplicativo.

`pure` debería recibir un valor de cualquier tipo y devolver un valor aplicativo que tiene ese valor como resultado, "dentro" de él. Es como envolver el valor `a` dentro de un valor aplicativo que tiene ese valor como resultado dentro de él.

Una mejor manera de pensar en `pure` sería decir que coge un valor y lo pone en algún tipo de contexto puro (un contexto mínimo que sigue produciendo ese valor).

La función `<*>` es muy interesante. Su declaración de tipos es:

```
f (a -> b) -> f a -> f b
```

Por tanto se parece bastante a `fmap :: (a -> b) -> f a -> f b`. Puedes pensar en la función `<*>` como un `fmap` concreto para funciones.

- `fmap` recibe una función y un valor de funtor y aplica la función dentro del valor del funtor.
- `<*>` coge un valor de funtor que tiene una función en él y otro funtor, extrae ese función del primer funtor y luego la mapea sobre el segundo.

Implementación de instancia Applicative para Maybe

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

Vemos que `f` juega el rol del funtor aplicativo, y debe recibir un tipo concreto como parámetro, así que escribimos `instance Applicative Maybe where` en vez de `instance Applicative (Maybe a) where`.

Después, tenemos `pure`. Recuerda que se supone que recibe algo y lo "envuelve" en un valor aplicativo. Se ha usado curryficación (ya que el constructor de valor `Just` es como una función normal)así que da lo mismo escribir eso que `pure x = Just x`.

La definición de `<*>` es muy descriptiva; en el caso de `Nothing` no hay ninguna función que obtener, y simplemente devuelve `Nothing`, en cambio en el caso del `Just f` con el reconocimiento de patrones obtiene la función `f`, para posteriormente hacer `fmap f something`.

¿Por qué no hemos tenido que preocuparnos del caso `(Just f) <*> Nothing`? Pues porque `fmap` sobre `Nothing` siempre devuelve `Nothing`.

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
```

Con los funtores normales, cuando mapeamos una función sobre un funtor, no podemos obtener el resultado de manera general, incluso si el resultado es una función parcialmente aplicada. Los funtores aplicativos, por otra parte, permiten operar sobre muchos funtores con una única función.

Estilo aplicativo

Con la clase de tipos `Applicative`, podemos encadenar el uso de la función `<*>`, lo que permite operar en varios valores aplicativos en vez de en uno.

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

Importante: la función `<*>` es *asociativa a izquierdas*.

```
pure (+) <*> Just 3 <*> Just 5
```

```
(pure (+) <*> Just 3) <*> Just 5
```

Primero la función `+` se pone en un valor aplicativo (en este caso un valor `Maybe` que contiene la función). Así que tenemos `pure (+)` lo cual es `Just (+)`.

Después, se aplica la función `Just (+) <*> Just 3`, que da `Just (3+)`, a causa de la curryficación, si a la función `+` sólo le aplicamos un `3` va a darnos una función que recibe un parámetro y le suma 3 a dicho parámetro.

Finalmente, se ejecuta `Just (3+) <*> Just 5`, lo cual da `Just 8`.

Por tanto, lo que conseguimos con el estilo aplicativo `pure f <*> x <*> y <*>` es coger una función que espera parámetros no aplicativos y usar esa función para operar en muchos valores aplicativos. La función puede recibir tantos parámetros como queramos porque se aplica parcialmente y uno por uno entre los sucesivos `<*>`.

`f <*> x` es lo mismo que `fmap f x`. Esta es una de las leyes aplicativas que veremos más adelante. Esto se generaliza de modo que `pure f <*> x <*> y <*>...` es lo mismo que `fmap f x <*> y <*>...`.

Por esto, el módulo `Control.Applicative` exporta una función llamada <$>, que es simplemente `fmap` como operador infijo. Está definida así:

```
(<\$>) :: (Functor f) => (a -> b) -> f a -> f b
f <\$> x = fmap f x
```

Usando <$> es como realmente se ve la potencia del estilo aplicativo ya que ahora:

- si queremos aplicar una función sobre tres valores aplicativos escribiremos: `f <$> x <*> y <*> z`.
- Si los parámetros eran valores normales en vez de funtores aplicativos escribiremos: f x y z.

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

Antes de ver qué ha ocurrido, comparemos la línea anterior con esta:

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

Volvamos a nuestro `(++) <$> Just "johntra" <*> Just "volta"`:

Primeramente, `<$>` es como hacer `fmap (++) (Just "johntra")`. Recuerda que `(++)` tiene el tipo `(++) :: [a] -> [a] -> [a]`.

Obtenemos entonces algo como `Just ("johntra"++)` lo cual tiene un tipo `Maybe ([Char] -> [Char])`. Date cuenta de que el primer parámetro de `(++)` fue "comido" y que las `a` cambiaron a `Char`.

Ahora se ejecutará `Just ("johntra"++) <*> Just "volta"`, que saca la función fuera del `Just` y la mapea sobre `Just "volta"`, lo cual da un resultado de tipo [Char].

Podrías pensar que el estilo aplicativo sólo funciona con `Maybes`. Mal hecho.

Listas

Las listas (realmente el constructor de tipo para listas, `[]`) son funtores aplicativos. Veámos cómo `[]` es instancia de `Applicative`.

```
instance Applicative [] where
pure x = [x]
fs <*> xs = [f x | f <- fs, x <- xs]
```

Recuerda: `pure` recibe un valor y lo pone en un contexto por defecto. En otras palabras, lo pone en el contexto mínimo que aún produce ese valor.

El contexto mínimo para las listas sería la lista vacía, sin embargo, la lista vacía representa la falta de valor, así que no podrá contener el valor que usamos al llamar a pure. Lo que se hizo en este caso es coger el valor recibido y devolver una lista singleton con el mismo.

Lo mismo ocurría con `Maybe`, el contexto mínimo sería `Nothing`, pero al dejar el valor con el que hemos llamado a `pure` fuera, se usa `Just`.

```
ghci> pure "Hola" :: [String]
["Hola"]
ghci> pure "Hola" :: Maybe String
Just "Hola"
```

Si la función `<*>` se limitara sólo a listas, tendría un tipo `(<*>) :: [a -> b] -> [a] -> [b]`. Está implementada con una compresión de listas. `<*>` debe, de alguna manera, extraer la función del parámetro izquierdo y luego mapearla sobre el parámetro derecho. En este caso, como el parámetro izquierdo es una lista, no sabemos de qué tamaño, lo que hace es coger la cabeza de la lista (una función) y mapearla sobre la lista parámetro derecho completa. Esto se repite hasta que la lista parámetro izquierdo sea vacía.

De modo que obtendremos una lista con todas las combinaciones posibles de aplicación de una función de la lista de la izquierda sobre los valores de la derecha.

```
ghci> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

Si tenemos una lista de funciones que reciba dos parámetros, podemos aplicar esas funciones entre dos listas:

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

De nuevo, hemos usado una función normal que recibe dos cadenas entre dos listas de cadenas, simplemente insertando los operadores aplicativos apropiados.

- Valores como `100` ó `"hola"` pueden ser vistos como computaciones deterministas que sólo devuelven un resultado.
- Valores como `[3,7,8]` pueden ser vistos como computaciones no deterministas que no saben a priori qué resultado tendrán. Así que si hacemos algo como `like (+) <$> [1,2,3] <*> [4,5,6]`, podemos pensar en ello como "juntar" dos computaciones no deterministas con `+`, para producir otra computación no determinista que está aún menos segura de su resultado.

Veamos un ejemplo:

```
ghci> (++) <\$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

El estilo aplicativo es muchas veces un buen reemplazo para compresiones de listas. Si quisiéramos obtener todos los productos posibles de los valores de dos listas, podríamos hacer:

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

Si pasamos esto al estilo aplicativo:

```
ghci> (*) <\$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

Esto es en cierto modo más claro, siempre que tengamos claro el estilo aplicativo.

Si queremos obtener todos los productos de las mismas listas que son mayores que 50, podemos hacer:

```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Es fácil ver que, para listas, `pure f <*> xs` es lo mismo que `fmap f xs`. `pure f` es `[f]`, y `[f] <*> x s` aplicará cada función de la lista izquierda sobre todos los valores de la lista derecha, con lo cual estamos mapeando `f`.

IO también es un funtor aplicativo

```
instance Applicative IO where
  pure = return
    a <*> b = do
    f <- a
    x <- b
    return (f x)
```

Recordemos que `pure` pone el valor que recibe en el contexto mínimo que sigue produciendo el mismo valor como resultado. Por tanto, tiene sentido que `pure` sea `return`. Recuerda que `return` devuelve una acción que no hace nada (salvo producir el valor que fue parámetro del `return`).

Si `<*>` fuera sólo para `IO`, tendría tipo:

```
(<*>) :: IO (a -> b) -> IO a -> IO b
```

En el caso de `IO`, recibe una acción de E/S `a`, la cual produce una función, ejecuta la función y liga esa función a `f`. Después ejecuta `b` y liga el resultado a `x`. Finalmente, ejecuta `f` con parámetro `x` .

Hemos usado sintaxis `do`, que, recordemos, une varias acciones E/S en una sóla.

Con `Maybe` y `[]`, podemos pensar en `<*>` como simplemente extraer una función de su parámetro izquierdo y aplicarla sobre su parámetro derecho.

Con `IO`, se sigue extrayendo, pero también introducimos la noción se *secuenciación*, porque estamos "pegando" dos acciones de E/S. Necesitamos extraer la función de la primera acción de E/S, pero para extraer un resultado de una acción de E/S, la acción de E/S debe ser llevada a cabo:

```
miAccion :: IO String
miAccion = do
  a <- getLine
  b <- getLine
  return $ a ++ b
```

Esta acción de E/S pedirá escribir dos líneas al usuario y producirá un resultado con las dos líneas concatenadas. Lo hemos conseguido mediante "pegar" dos acciones de E/S en otra que produce el resultado `a ++ b`. Se puede hacer mediante el estilo aplicativo así:

```
miAccion :: IO String
miAccion = (++) <$> getLine <*> getLine
```

Recordemos que el tipo de `getLine` es `getLine :: IO String`. Cuando usamos `<*>` entre dos valores aplicativos, el resultado es un valor aplicativo, luego todo tiene sentido.

`getLine` es una "caja" que va al mundo real a traernos una cadena. LLamar a `(++) <$> getLine <*> getLine` produce una *nueva* acción de E/S que envía dos cajas al mundo real a traernos líneas del terminal y luego nos da la concatenación de esas dos líneas como resultado.

Veamos qué tipo produce nuestra nueva acción:

```
Prelude> import Control.Applicative
Prelude Control.Applicative> :t (++) <$> getLine <*> getLine
(++) <$> getLine <*> getLine :: IO [Char]
```

```
main = do
  a <- (++) <$> getLine <*> getLine
  putStrLn $ "La concatenación de las dos líneas es: " ++ a
```

Funciones como aplicativos:

Otra instancia de `Applicative` es `(->) r`, o funciones.

```
instance Applicative ((->) r) where
  pure x = (\_ -> x)
  f <*> g = \x -> f x (g x)
```

Recuerda: a consecuencia de la curryficación, la aplicación de funciones se asocia a izquierdas.

Cuando envolvemos un valor en un valor aplicativo con `pure`, el resultado que produce debe ser ese valor. Un contexto mínimo debe producir ese valor, por ello devolvemos una función (lambda) que ignorará su parámetro y devolverá el valor con el cual fue creada.

El tipo de `pure` para `(->) r` es:

```
pure :: a -> (r -> a)

ghci> (pure 3) "blah"
3
```

Debido a que la aplicación de funciones es asociativa a izquierdas, podemos omitir los paréntesis:

```
ghci> pure 3 "blah"
3

ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <\$> (+3) <*> (*100) $ 5
508
```

LLamar a `<*>` con dos valores aplicativos devuelve un valor aplicativo, así que si lo usamos sobre dos funciones dará una función.

Cuando hacemos `(+) <\$> (+3) <*> (*100)`, estamos creando una función que usará `+` en los resultados de (+3) y (*100) y devuelve el resultado de la suma. Con `(+) <\$> (+3) <*> (*100) $ 5`, `(+3)` y `(\*100)` son aplicados primero a `5`, dando como resultados `8` y `500`. Luego `+` es llamado con `8` y `500`, dando como resultado `508`.

El siguiente código es similar:

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

Creamos una función que hará `\x y z -> [x,y,z]` con los eventuales resultados de `(+3)`, `(*2)` y `(/2)`.

Nota: no es demasiado importante saber cómo funciona la instancia de `(-> r)` para `Applicative`. Lo que realmente importa es aprender a usar funciones al estilo aplicativo.

Zip con listas

Si escribimos `[(+3),(*2)] <*> [1,2]`, `(+3)` se aplicará al `1` y al `2`, y `(*2)` también se aplicará al `1` y al `2`, dando una lista con cuatro elementos: `[4,5,2,4]`.

Sin embargo, `[(+3),(*2)] <*> [1,2]` podría funcionar de modo que la primera función se aplicara al primer valor, la segunda al segundo valor, etc. Lo cual daría una lista con dos valores: `[4,4]`. Puede verse como `[1 + 3, 2 * 2]`.

Para conseguir esto, en el módulo `Control.Applicative` existe la instancia `ZipList`, que se añadió porque un tipo no puede tener dos instancias de la misma clase de tipos. `ZipList` tiene un constructor con un único campo (una lista):

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

Por tanto, `<*>` aplica la primera función al primer valor, la segunda función al segundo valor, etc. Esto se consigue con `zipWith (\f x -> f x) fs xs`. Debido al comportamiento de `zipWith`, la lista resultado tendrá el largo de la más corta de las dos.

Esto nos va a dar una pista de por qué se ha hecho que `pure x = ZipList (repeat x)`.

Recuerda: `repeat` devuelve una lista infinita con todos sus elementos valiendo el valor que le pasemmos a `repeat`.

Esto es así porque así se aplicará la función que queramos sobre *toda* la lista objetivo y no sólo sobre el primer elemento. Esto satisface la ley que dice `pure f <*> xs` es igual a `fmap f xs`.

Si `pure 3` sólo devolviese `ZipList [3]`, `pure (*2) <*> ZipList [1,5,10]` devolvería `ZipList [2]`, porque la lista resultante del `zip` de dos listas coincide con la de menor tamaño.

getZipList

Como `ZipList` no es instancia de `Show`, si queremos imprimir sus valores debemos extraer una lista "normal" de una `ZipList`. Para ello existe la función `getZipList`.

```
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Nota: la función `(,,)` es la misma que `\x y z -> (x,y,z)`. Asimismo, la función `(,)` es la misma que `\x y -> (x,y)`.

Además de `zipWith`, la librería estándar tiene `zipWith3`,`zipWith4`,..., hasta `zipWith7`. `zipWith` recibe una función con dos parámetros y "zipea" dos listas con ella. `zipWith3` recibe una función con tres parámetros y "zipea" tres listas con ella, etc. Pero usando `ZipList` con estilo aplicativo no necesitamos tener muchas funciones distintas según el número de listas que queramos procesar, podemos "zipear" un número arbitrario de listas con una función, lo cual es bastante cómodo.

Leyes de Applicative

La primera (vista anteriormente) es la más importante, aunque las otras tienen su importancia también.

- `f <*> x = fmap f x`
- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

Funciones útiles para Applicatives

`Control.Applicative` define una función llamada `liftA2`, la cual tiene el tipo:

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
```

Se define así:

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Simplemente aplica una función entre dos aplicativos, ocultando el estilo aplicativo.

Con funtores ordinarios, sólo podemos mapear funciones sobre un valor de funtor. Con funtores aplicativos, podemos aplicar una función sobre muchos valores de funtor. Sería interesante ver el tipo de `liftA2` como:

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

Luego podemos decir que `liftA2` es una función que recibe una función binaria normal y la "promociona" a una función que opera sobre dos aplicativos.

Podemos recibir dos valores aplicativos y combinarlos en un valor aplicativo que tenga dentro de él los resultados de esos dos valores aplicativos en una lista. Por ejemplo, tenemos `Just 3` y `Just 4`. Asumamos que el segundo contiene una lista singleton:

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

Ahora tenemos `Just 3` y `Just [4]`. ¿Cómo obtenemos [3,4]?

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

`(:)` es una función que recibe un elemento y una lista y devuelve una nueva lista con ese elemento al principio. Ahora que tenemos `Just [3,4]`, ¿podríamos combinar eso con `Just 2` para obtener [2,3,4]? Sí, podríamos combinar tantos valores aplicativos como queramos en una lista que contiene los resultados de esos valores aplicativos dentro de la misma.

Implementemos una función que reciba una lista de valores aplicativos y devuelva un valor aplicativo que tenga una lista como su valor resultado:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

Primero, miremos el tipo; transformará una lista de valores aplicativos en un valor aplicativo con una lista. Si queremos poner una lista vacía en un valor aplicativo con una lista de resultados, simplemente ponemos la lista vacía en un contexto por defecto.

Si tenemos una lista con una cabeza y una cola (x es un valor aplicativo y xs una lista de ellos), llamamos a `sequenceA` en la cola, lo cual resulta en un valor aplicativo con una lista dentro. Luego simplemente insertamos por el principio el valor dentro del aplicativo con una lista, y ya está.

```
sequenceA [Just 1, Just 2]

(:) <$> Just 1 <*> sequenceA [Just 2]

(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])

(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])

(:) <$> Just 1 <*> Just [2]
```

Lo cual es lo mismo que `Just [1,2]`.

Recuerda: todas las funciones en las cuales recorremos una lista elemento por elemento y acumulamos un resultado por el camino pueden ser implementadas con un fold:

Podemos implementar `sequenceA` con un fold:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

Veamos algunos ejemplos:

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

No te preocupes si no has entendido el último ejemplo, será explicado un poco más adelante.

Recuerda: la función `and` devuelve `True` si todos los elementos tipo `Bool` de una lista son `True`, y `False` en cualquier otro caso. `any` es análoga pero parecida a la puerta lógica OR, es decir, devuelve `True` desde que un sólo elemento de la lista valga `True`.

Tenemos un número y queremos saber si cumple todos los predicados de una lista:

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

También podemos hacerlo con `sequenceA`:

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```

`sequenceA [(>4),(<10),odd]` crea una función que recibirá un número y le aplicará todos los predicados que tenemos en `[(>4),(<10),odd]`, devolviendo una lista de `Bool`.

Convierte una lista con tipo `(Num a) => [a -> Bool]` en una función con el tipo `(Num a) => a -> [Bool]`.

Como las listas son homogéneas, todas las funciones de la lista deben tener el mismo tipo. No podríamos tener una lista como `[ord, (+3)]`, poruqe `ord` recibe un carácter y devuelve un número, y `(+3)` recibe un número y devuelve un número. Cuando se usa con `[]`, `sequenceA` recibe una lista de listas y devuelve una lista de listas. Realmente crea listas que tienen todas las combinaciones posibles de sus elementos.

Aclaremos un poco aquel ejemplo que no se entendía a priori haciéndolo primero con `sequenceA` y después con una compresión de listas:

```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
Applicativghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2],[3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> sequenceA [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
```

`(+) <\$> [1,2] <*> [4,5,6]` da una computación no determinista `x + y`, donde `x` toma todos los valores en `[1,2]` e `y` toma todos los valores en `[4,5,6]`. Representamos eso como una lista que contiene todos los resultados posibles. De modo análogo, cuando hacemos `sequenceA [[1,2],[3,4],[5,6]]`, el resultado es una computación no determinista `[x,y,z]`, donde `x` toma todos los valores en `[1,2]`, `y` toma todos los valores en `[3,4]` etc. Para representar los valores de esa computación no determinista usamos una lista donde cada elemento de la lista es una posible lista. Por eso el resultado es una lista de listas.

Cuando se usa sobre acciones de E/S, `sequenceA` es lo mismo que `sequence`, recibe una lista de acciones de E/S y devuelve una acción E/S que hará cada una de esas acciones y tendrá como resultado una lista con los resultados de cada una de esas acciones de E/S.

Eso es porque para convertir un valor `[IO a]` en un valor `IO [a]`, para hacer una acción de E/S que produzca una lista de resultados cuando se ejecute, todas esas acciones de E/S deben ser secuenciadas para que se ejecuten una detrás de otra cuando la evaluación sea forzosa. No se puede obtener el resultado de una acción E/S sin llevarla a cabo.

```
ghci> sequenceA [getLine, getLine, getLine]
ola
k
ase
["ola","k","ase"]
```

Como hemos visto, simplemente usando `<$>` y `<*>`, podemos emplear funciones normales para operar uniformemente en cualquier número de funtores aplicativos y aprovechar las ventajas de cada uno.

Kinds

Los kinds son el "tipo de los tipos". Los tipos son pequeñas etiquetas que los valores llevan consigo así que de ese modo podemos razonar sobre sus valores. Pero los tipos tienen sus propias pequeñas etiquetas que se llaman kinds.

Nota: el comando de GHCi `:k` nos permite comprobar el kind de un tipo.

```
Prelude> :k Int
Int :: *
```

¿Qué significa esa `*`? Indica que el tipo es un tipo concreto. Un **tipo concreto** es un tipo que que no recibe parámetros de tipo. Los valores **sólo** pueden tener tipos que sean tipos concretos.

Veamos cuál es el tipo de Maybe:

```
Prelude> :k Maybe
Maybe :: * -> *
```

Este kind nos dice que el constructor de tipos `Maybe` recibe un tipo concreto (como `Int`) y devuelve otro tipo concreto (como `Maybe Int`). Como `Int -> Int` significa que una función recibe un `Int` y devuelve un `Int`, `* -> *` significa que el constructor de tipo recibe un tipo concreto y devuelve un tipo concreto. Apliquémosle un parámetro a `Maybe` y veamos qué tipo nos queda:

```
Prelude> :k Maybe Bool
Maybe Bool :: *
```

Un paralelismo es que, aunque kinds y tipos son dos cosas diferentes, se comportan de manera parecida, ya que usan la curryficación; por ejemplo: `:t isUpper` y `:t isUpper 'A'`. La función `isUpper` tiene tipo `a -> Bool`, mientras que `:t isUpper 'A'` tiene tipo `Bool`. Sin embargo, si atendemos a los kinds de estos tipos:

```
Prelude Data.Char> :k Char -> Bool
Char -> Bool :: *
Prelude Data.Char> :k Bool
Bool :: *
```

Veamos ahora el tipo de `Either`:

```
Prelude> :k Either
Either :: * -> * -> *
```

Esto nos dice que `Either` recibe dos tipos concretos como parámetros de tipo para producir un tipo concreto. También parece la declaración de tipos de una función que recibe dos valores y devuelve algo. Los constructores de tipos están curryficados (como las funciones), así que podemos aplicarlos parcialmente:

```
Prelude> :k Either String
Either String :: * -> *
Prelude> :k Either String Int
Either String Int :: *
```

Cuando quisimos hacer `Either` parte de la clase de tipos `Functor`, necesitamos aplicarla parcialmente, porque `Functor` sólo quiere tipos con kind `* -> *`, por tanto necesitamos aplicar parcialmente `Either` para obtener esto en vez de su kind original `* -> * -> *`.

Si volvemos a mirar la definición de `Functor`, vemos que la variable de tipo `f` se usa como un tipo que recibe un tipo concreto para producir un tipo concreto.

```
class Functor where
  fmap :: (a -> b) -> f a -> f b
```

Sabemos que debe producir un tipo concreto, porque se usa como el tipo de un valor en una función. Y por ello deducimos que los tipos que quieren ser amigos de `Functor` deben tener un kind `* -> *`.

Mónadas

`Monad` es una clase como otra cualquiera (sí, aunque no te lo creas).

Nota: para que un tipo pueda ser instancia de `Monad` tiene que tener kind `* -> *`. Este es un requisito **necesario, pero no suficiente**.

- `Maybe :: * -> *`
- `Maybe a :: *` (ahora ya (Maybe a) no puede ser aplicado a ningún tipo más).
- `Either :: * -> * -> *`
- `Either String :: * -> *`

El tipo tiene que tener kind `* -> *` para poder ser instancia de Monad. Tener un parametro libre, digamos `Bool` no podría, `Either`tampoco, `Either Bool` sí.

Lo mismo pasa con la clase `Functor`.

Los métodos de `Monad` son:

```
(>>= ) :: m a -> (a -> m b) -> m b
return :: a -> m a
```

donde `a`, `b` son cualquier cosa y `m` es la mónada que estamos instanciando, el tipo que estamos instanciando en la clase de tipos `Monad`.

Cuando escribes la instancia, como cuando escribes muchas funciones en Haskell, el tipo te va a forzar a escribir lo único que tiene sentido escribir. Definamos un tipo que, aunque en apariencia inútil, tiene sus usos prácticos, el tipo identidad:

```
data Id a = Id a
```

En la práctica, dado un tipo `a`, los elementos que puedes construir en `a` y los que puedes construir en `Id a` son prácticamente los mismos por eso se llama identidad.

`Id` puede ser instanciado en la clase `Functor`.

Recuerda: sólo hay un método en la clase Functor `fmap :: (a -> b) -> f a -> f b` donde `f` es el tipo que estamos instanciando, el resto es polimórfico.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Como se ve, en la definición aparece `Functor f` es en ese momento donde se especifica: en los métodos que voy a describir abajo, `f` es el tipo de cada instancia de esta clase entonces, para hacer una instancia la clase, hay que escribir una definición de cada método, sustituyendo el parámetro `f` (en este caso) con el que estamos instanciando.

El kind que hace falta lo calcula GHC a partir de la definición de la clase. En fmap se puede ver que `f` está siendo aplicado a un tipo. De ahí deduce que el kind tiene que ser `* -> *`.

Entonces, para hacer Id instancia de Functor necesitamos definir `fmap :: (a -> b) -> Id a -> Id b`.

Importante: el operador `(->)`, en ausencia de paréntesis, asocia a la derecha, osea que `(a -> b) -> Id a -> Id b` significa `(a -> b) -> (Id a -> Id b)` y NO `((a -> b) -> Id a) -> Id b`.

Por tanto, se puede construir la equivalencia:

```
(a -> b) -> Id a -> Id b = (a -> b) -> (Id a -> Id b)
```

Es decir, estamos **elevando** (lifting) una función que opera sobre tipos a otra que opera sobre funtores `Id`. Es decir, esta función toma elementos en el funtor y los devuelve en el funtor también.

Importante: todo en Haskell trata de hacer encajar los tipos:

```
f :: a -> b
x :: a
```

Por tanto:

```
f x :: b
```

Recordemos que en nuestra sustitución en el tipo generalista de `fmap`, `fmap :: (a -> b) -> Id a -> Id b`, queremos que `fmap` devuelva un resultado de tipo `Id b`, luego nuestra definición de `fmap` no puede ser otra que:

```
fmap f x = Id (f x)
```

Dado que:

```
Id :: a -> Id a
```

Ahora que `Id` forma parte de la clase `Functor`, hagamos a este tipo instancia de la clase de tipos `Monad`:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Para ello, debemos hacer lo mismo que hicimos con `Functor`, pero en este caso debemos definir dos funciones, `(>>=)`, que se conoce como bind, y `return`, que, recordemos, tiene muy poco que ver con los returns de otros lenguajes como C.

Empecemos por lo fácil, definamos `return`.

```
return :: a -> Id a
return = Id
```

Ya que antes vimos que `Id :: a -> Id a`.

Ahora definamos `(>>=)`:

Su tipo general es: `(>>=) :: Monad m => m a -> (a -> m b) -> m b`.

Su tipo concreto para el tipo `Id a` es:

```
(>>=) :: Id a -> (a -> Id b) -> Id b
```

Luego la propia definición nos programa a nosotros y nos dice qué tenemos que programar; como el primer argumento es de tipo `Id a`, tenemos que hacer reconocimiento de patrones (pattern matching):

```
(>>=) :: Id a -> (a -> Id b) -> Id b
(Id a) >>= f = f a
```

Todo encaja. Mediante pattern matching hemos "sacado" el argumento tipo `a` de `Id a`, hemos aplicado `f` a dicho argumento y se procude al final un valor de tipo `Id b`, ¡luego hemos definido nuestra primera mónada!

```
(>>) :: m a -> m b -> m b
```

Por defecto se define como

```
(>>) :: m a -> m b -> m b
m >> k = m >>= \_ -> k
```

Así que es sólo para ahorrate la lambda que ignora su argumento.

Digamos que `m :: m a`, para alguna instancia de `Monad m` la función de la derecha es constantemente `k` así que el valor en `m` realmente es ignorado y devuelve `k`.

Hace un bind, pero con una función constante que no depende de la `a` digamos.

`(>>)` **no** es simplemente ignorar el primer argumento. Lo hacemos por definición:

```
m >> k = m >>= \_ -> k
```

¿Qué resultado dará la siguiente expresión?

```
(Id 3 >> Id 4)
```

Aplicamos definición:

```
Id 3 >>= \_ -> Id 4
```

Como vemos, se transforma en un bind, y por la definición de bind tenemos:

```
(\_ -> Id 4) 3
```

Lo cual ignora el parámetro y devuelve `Id 4` como resultado. Es importante remarcar que **no todas las mónadas** ignoran su primer parámetro y devuelven el segundo.

Notación do

La notación `do` lo que hace es transformar:

```
a <- m
e
```

en `m >>= \a -> e` y

```
m
n
```

en `m >> n`.

Veamos un ejemplo:

```
x <- Id 3
return x
```

Se convierte en:

```
Id 3 >>= \x -> return x
```

Ahora intentemos seguir, sustituyendo la función bind por su definición:

```
(\x -> return x) 3
```

```
return 3
```

```
Id 3
```

Otro ejemplo, un poco más grande:

```
x <- Id 2
y <- Id 3
return x
```

Quedaría:

```
Id 2 >>= (\x -> Id 3 >>= (\y -> return x))
```

Que se puede escribir de diversas maneras para comprenderlo mejor o aumentar la legibilidad:

```
Id 2 >>= (\x ->
  Id 3 >>= (\y ->
    return x))
```

Podríamos interpretarlo como que el valor 2 se ligó a la `x`, el valor 3 a la `y` y al final devolvimos el primero, el ligado a x.

Recuerda: el constructo `<-` "saca" de un contexto monádico los valores contenidos en dicho contexto. Es decir, si tenemos una mónada `M a`, donde `a` puede ser cualquier tipo, `<-` sobre esa mónada nos devolverá el valor `a` tal cual, sin contexto.

Importante: si un valor se liga a un nombre, podremos usar ese nombre en toda la expresión `do` (aunque **sólo** será usable dentro del `do`.

Norma sintáctica: el último término de una expresión `do` debe tener tipo `m a` para algún tipo `a`. Aquí `m` representa una mónada.

Recordemos las reglas de transformación de expresiones `do`:

```
a <- m
e
```

en `m >>= \a -> e` y

```
m
n
```

en `m >> n`.

Como vemos, los valores `e` y `n`, es decir, los resultados, con cosas de tipo `m a`.

Como ejemplo, se muestran tres funciones equivalentes pero escritas en distinta notación:

```
seqnCases :: Maybe a -> Maybe b -> Maybe (a,b)
seqnCases a b = case a of
                  Nothing -> Nothing
                  Just x -> case b of
                              Nothing -> Nothing
                              Just y -> Just (x,y)

seqnDo :: Maybe a -> Maybe b -> Maybe (a,b)
seqnDo a b = do x <- a
                y <- b
                Just (x,y)

seqnBind :: Maybe a -> Maybe b -> Maybe (a,b)
seqnBind a b = a >>= \x -> b >>= \y -> Just (x,y)
```

La mónada Maybe

`Maybe` tiene kind `Maybe :: * -> *`, lo que nos hace sospechar que quizá pueda hacerse instancia de `Functor` y `Monad`.

La definicion del tipo `Maybe` es:

```
data Maybe a = Just a | Nothing
```

Vemos que es un tipo suma, pues puede tener valores de diferentes "formas".

Tiene dos constructores de datos:

```
Just :: a -> Maybe a
Nothing :: Maybe a
```

Algo interesante sobre `Nothing` es que habita muchos tipos:

```
Nothing :: Maybe Int
Nothing :: Maybe Bool
Nothing :: Maybe (Maybe Int)
```

Y muchos más. La clave de esto es que **no** todos los `Nothing` son iguales.

Vamos a hacer `Maybe` instancia de `Functor`, en este caso, `fmap :: (a -> b) -> Maybe a -> Maybe b`:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap _ Nothing  = Nothing
fmap f (Just x) = Just (f x)
```

Como tenemos dos constructores, nos han hecho falta dos ecuaciones (o una ecuación si hubiéramos usado case, pero bueno, dos pattern matchings en fin y al cabo).

Como sólo tenemos un argumento `Maybe`, hay que hacer una ecuación por constructor.

Si hubieran dos argumentos `Maybe`, tendríamos que hacer 4. Siempre se pueden reducir los casos, y como mucho serían 4.

Así que `fmap` en el tipo `Maybe`, aplica la función `f` a lo que tenga dentro si es que tiene...el `Nothing` lo deja tal cual aunque lo cambia un poquito, le cambia el tipo! lo pasa de `Nothing :: Maybe a` a `Nothing :: Maybe b`, osea que en realidad no son el mismo `Nothing`.

Veamos si podemos hacer la instancia de `Monad`, empezamos con `return`, nuestras herramientas son:

- Just :: a -> Maybe a
- Nothing :: Maybe a
- El pattern matching

La declaración de tipos más general de `return` es:

```
return :: a -> m a
```

Por tanto podríamos definir:

```
return :: a -> Maybe a
return = Just
```

¡Parece que sale sólo! Como hemos dicho, los tipos de las funciones nos obligan a escribir casi siempre lo que se *debe* escribir.

Como no hay ningún argumento de tipo `Maybe` no hay que hacer pattern matching. Además, los tipos encajan, lo cual es imprescindible.

Ahora, implementemos `(>>=)`:

Recuerda, el tipo más general de `(>>=)` es:

```
(>>=) :: M a -> (a -> M b) -> M b
```

Por tanto, si pasamos eso a nuestra futura mónada `Maybe`:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing (>>=) _  = Nothing
(Just x) (>>=) f = f x
```

Por tanto, nuestra mónada que era futura ahora es presente, !Ya tenemos otra mónada!

Esta mónada es algo más interesante, porque si nos damos cuenta, si miramos la definición del bind para `Nothing` vemos que no importa la función que tenga a la derecha, siempre dará `Nothing`. Esto tendrá sus consecuencias.

Sabemos que, en general:

```
(>>) :: M a -> M b -> M b
m >> k = m >>= \_ -> k
```

Pero, ¿cómo funciona `(>>)` en el caso del tipo `Maybe`? Sustituyamos ese `(>>)` por el de la instancia `Maybe` para averiguarlo.

```
(>>) :: Maybe a -> Maybe b -> Maybe b
m >> k = m >>= \_ -> k
```

Vayamos un poco más allá y veamos ahora qué pasa si sustituímos el operador bind:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing (>>=) _  = Nothing
(Just x) (>>=) f = f x
```

Luego, de `m >> k = m >>= \_ -> k` nos quedamos con:

```
m >>= \_ -> k
```

Que se sustituye por:

```
Nothing >>= _ = Nothing
(Just x) >>= (\_ -> k) = (\_ -> k) x
                       = k
```

Por tanto, si un `Nothing` está a la izquierda del operador `(>>)`, el resultado será Nothing, sin importar lo que haya a la derecha.

Para el caso de `Just x`, acabaremos devolviendo su segundo argumento, `k`.

Para simplificar un poco:

```
case m of
  Nothing -> Nothing
  _ -> k
```

La función `div` es la división entera:

```
Prelude> div 4 2
2
Prelude> div 7 2
3
```

Lo malo es que si intentamos dividir algo por cero:

```
Prelude> div 7 0
*** Exception: divide by zero
```

Veamos el tipo de `div`:

```
div :: Integral a => a -> a -> a
```

El tipo es el gran problema de la función `div`, pues obliga a que el resultado sea del mismo tipo que dividendo y divisor. Esto deja a la función sin opciones para el caso en el que el divisor es 0.

Un tipo más realista para `div` sería:

```
div :: Int -> Int -> Maybe Int
```

Lo malo de usar un tipo de retorno `Maybe Int` es que tenemos que usar pattern matching para ver qué hacemos en cada caso:

```
gooddiv :: Int -> Int -> Maybe Int
gooddiv _ 0 = Nothing
gooddiv n m = Just (div n m)
```

Ahora queremos hacer varias divisiones, y devolver la suma de los resultados si ambas divisiones han tenido éxito. Con el antiguo `div` haríamos:

```
myop a1 b1 a2 b2 = div a1 b1 + div a2 b2
```

Lo malo de esto es que puede dar excepciones si `b1` y/o `b2` valen 0.

Así que vamos a usar `gooddiv`. El problema es que no puedo usar `(+)` con el tipo Maybe de por medio, la siguiente definición daría error de tipos:

```
myop a1 b1 a2 b2 = gooddiv a1 b1 + gooddiv a2 b2
```

Para arreglar esto, deberíamos hacer pattern matching en los resultados:

```
myop a1 b1 a2 b2 = case gooddiv a1 b1 of
  Nothing -> Nothing
  Just c1 -> case gooddiv a2 b2 of
    Nothing -> Nothing
    Just c2 -> Just (c1 + c2)
```

Quizá este esquema podría servir para funciones sencillas como `myop` (a pesar de ser un poco tedioso de programar). Sin embargo, esta sección del tutorial trata de mónadas, lo cual nos hace pensar que quizás nos puedan ayudar.

Claro, ¡`Maybe` es instancia de `Monad`! ¿Qué pasa si hago esto?

```
do c1 <- gooddiv a1 b1
   c2 <- gooddiv a2 b2
   return (c1 + c2)
```

Recordemos las reglas de transformación de expresiones `do`:

```
a <- m
e
```

en `m >>= \a -> e` y

```
m
n
```

en `m >> n`.

Hagamos uso de las reglas de traducción para pasar esto a notación usando bind:

```
gooddiv a1 b1 >>= (\c1 ->
gooddiv a2 b2 >>= (\c2 ->
return (c1 + c2)))
```

Esto se puede leer como: aplica `gooddiv` a `a1 b1` y llama a su valor resultado `c1`, luego aplica `gooddiv` a `a2 b2` y llama a su valor resultado `c2`, y por último combina los dos resultados usando la función `(+)`.

Recordemos la definición de bind para `Maybe`:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing (>>=) _  = Nothing
(Just x) (>>=) f = f x
```

Ahora usando esta información vamos a "traducir" lo siguiente:

```
gooddiv a1 b1 >>= (\c1 ->
gooddiv a2 b2 >>= (\c2 ->
return (c1 + c2)))
```

Notar que las dos ecuaciones de `(>>=)` se ven como un `case` que puede dar: 1) `Nothing` ó 2) Un resultado en un nombre.

```
case gooddiv a1 b1 of
  Nothing -> Nothing
  Just c1 -> case gooddiv a2 b2 of
    Nothing -> Nothing
    Just c2 -> return Just (c1 + c2)
```

¡Por tanto esto es equivalente a la función que hicimos antes usando `cases`!

Por tanto, como vemos, la mónada nos evita toda esa escritura de casos, ya que ella misma los maneja mediante sus definiciones. A parte de ello, desde que a la izquierda de un bind (o a la derecha en la notación `do`) haya un `Nothing`, ahorraremos gran cantidad de evaluación (todo lo que quede, el resultado siempre será `Nothing`).

Por tanto, la mónada `Maybe` permite:

- Ahorrar mucho pattern matching, todo se maneja como si fuera a funcionar. Si no funciona, los métodos de la mónada actuarán en consecuencia.
- Ahorrar tiempo de evaluación en los casos donde se sabe qué resultado dará una expresión.

Caso de uso de la mónada `Maybe`:

Cuando queremos encadenar varias operaciones que pueden "fallar" y queremos usar sus resultados, es el momento perfecto.

Expresión típica del operador >>= para Maybe

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
mn >>= \xn ->
f x1 x2 ... xn
```

Lo cual significa, evalúa cada expresión `m1`, `m2`,…,`mn` una por una, y luego combina sus valores resultado `x1`, `x2`,…,`xn` mediante la aplicación de `f`. La definición de `(>>=)` asegura que una expresión así sólo tiene éxito (devuelve un valor creado con `Just`) si toda mi en la secuencia tiene éxito. Dicho de otro modo, el programador no se tiene que preocupar por los posibles fallos (devolver `Nothing`) de ningún componente de la expresión, ya que `(>>=)` se encarga de ello automáticamente.

Lo mismo, pero expresado mediante notación `do`:

```
do x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

liftM2

En el módulo `Control.Monad` se encuentra la función `liftM2`, que tiene tipo:

```
liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
```

Esto se parece bastante a `fmap` pero con dos argumentos. Ahora recordemos la primera definición que hicimos de `myop`:

```
myop a1 b1 a2 b2 = div a1 b1 + div a2 b2
```

Si hacemos el operador prefijo:

```
myop a1 b1 a2 b2 = (+) (div a1 b1) (div a2 b2)
```

Por tanto podemos usar `liftM2`:

```
mygoodop a1 b1 a2 b2 = liftM2 (+) (gooddiv a1 b1) (gooddiv a2 b2)
```

Por tanto hemos escrito una función que maneja perfectamente la posibilidad de fallos que implican las divisiones por cero ¡en una sola línea!

El operador `(>>=)` evita el problema de las tuplas anidadas de resultados porque el resultado del primer argumento está directamente disponible para ser procesado por el segundo. Por tanto, `(>>=)` integra la secuenciación de valores de tipo `Maybe` con el procesamiento de sus valores resultado. Se le llama *bind* porque el segundo argumento enlaza el resultado del primero.

Es importante también que `(>>=)` es (en el caso de `Maybe`) simplemente la función `aplicar` con el orden de los argumentos invertido.

```
aplicar :: (a -> Maybe b) -> Maybe a -> Maybe b
aplicar _ Nothing  = Nothing
aplicar f (Just x) = f x
```

La mónada lista

```
instance Monad [] where
  return :: a -> [a]
  return x = [x]

  (>>=)  :: [a] -> (a -> [b]) -> [b]
  xs >>= f = concat (map f xs)
```

Ahora presentaré tres maneras de definir una función `pairs` que produce todas las tuplas de dos elementos posibles dadas dos listas:

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x, y)
```

Veamos si es cierto eso, desconfía siempre de la gente, !sobre todo de mí!

```
*Main> pairs [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

Ahora traduzcamos, mediante nuestras reglas de oro, la notación `do`:

```
pairs' :: [a] -> [b] -> [(a,b)]
pairs' xs ys = xs >>=
                 (\x -> ys >>=
                   (\y -> return (x,y)))

*Main> pairs' [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

Ahora expandiremos la definición de `pairs` para ver el despliegue de `(>>=)` para la mónada lista:

```
xs >>= (\x -> ys >>= (\y -> return (x,y)))                => por la definición de bind
concat (map (\x -> ys >>= (\y -> return (x,y)))) xs       => por la definición de bind
concat (map (\x -> concat (map (\y -> return (x,y)) ys)) xs)

*Main> concat (map (\x -> concat (map (\y -> return (x,y)) [3,4])) [1,2])
[(1,3),(1,4),(2,3),(2,4)]
```

Esto se parece mucho a una comprensión de listas, por lo cual podríamos sospechar que en realidad la compresión de listas es azúcar sintáctico para el uso de la mónada lista:

```
pairs'' xs ys = [(x,y) | x <- xs, y <- ys]
```

¿Miente el creador del tutorial? En la mayoría de los casos, la respuesta es sí, pero esta es una de las excepciones que confirman la regla.

```
*Main> pairs'' [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

La mónada escritora

Esta sección necesita mejoras:

Veamos ahora una mónada bastante interesante, que permite ir guardando la información histórica que queramos.

```
data Writer m a = Writer m a

class ForWriter t where
  something :: t
  combine :: t -> t -> t
```

Hagamos las instancias por orden, para cumplir los requisitos de GHC 7.10, debemos hacer todas nuestras mónadas instancia de `Applicative`, pero para que algo sea instancia de `Applicative` primero debe ser instancia de `Functor`:

Recuerda: `fmap :: Functor f => (a -> b) -> f a -> f b`.

```
instance Functor (Writer m) where

  fmap f (Writer m a) = Writer m (f a)
```

Recuerda: `pure :: a -> f a (<*>) :: f (a -> b) -> f a -> f b (<*>) :: Writer m (a -> b) -> Writer m a -> Writer m b`

```
instance ForWriter m => Applicative (Writer m) where

  pure a = Writer something a

  Writer m f <*> Writer m' a = Writer (combine m m') (f a)
```

Finalmente, hagamos una mónada que compile:

Recuerda: `return :: a -> m a (>>=) :: m a -> (a -> m b) -> m b`

```
instance ForWriter m => Monad (Writer m) where

  return a = Writer something a

  Writer m a >>= f =
    let Writer m' b = f a -- esto se puede hacer porque el tipo sólo tiene un constructor
      in Writer (combine m m') b
```