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

Introducción

Todo el código de este tutorial ha sido testeado con GHC 7.8.4 y 7.10.1. De todas formas, si he cometido algún error, ruego que me lo comuniques a la sección Issues del repositorio del tutorial.

Referencias indispensables

Informe oficial revisado de Haskell 98: nos permite ver la implementación que en su momento se hizo de las funciones predefinidas. Es muy importante la sección del módulo `Prelude`. Puedes entrar aquí para ver el índice completo.

Haskell Hierarchical Libraries: podemos ir al módulo que queramos y una vez dentro de él ver las definiciones de las funciones. Además, pulsando el botón *source* podemos ver el código real de las mismas, y aprender mucho.

reddit: un sitio impresionante, muy actualizado y completo, en el que te enterarás de muchas novedades y proyectos interesantes en Haskell, así como de tutoriales, vídeos y mucho más.

dohaskell.com: contiene enlaces muy interesantes a tutoriales, vídeos, conferencias, documentación…etc. Imprescindible.

Consejos para el lector

1. Si no entiendes algo, probablemente creas que entiendes todo lo anterior, sin ser esto cierto. Repasa y fortalece tus conocimientos.
2. Si no entiendes algo a la primera, sigue adelante (o aplica el punto 2) y retómalo un poco más tarde.
3. El mejor medio para aprender no es este tutorial, es Internet. En Internet tienes muchísima información actualizada minuto a minuto, ¡úsala! (ver “Referencias Indispensables”).
4. Pasa tu primera etapa de aprendizaje resolviendo problemas de juguete, más adelante, pasa a programas del mundo real.
5. Si alguna vez te aburres de leer tutoriales y/o libros, pasa a programar, escribe código que te haga ver utilidad a lo que estás aprendiendo.

Estado del tutorial

- [x] Sintaxis resaltada con colores
- [x] Ortografía
- [x] Pruebas del código incluido
- [] Reorganización del contenido
- [ ] Completado

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.

Mi sistema favorito para programar en Haskell es GNU/Linux, y en concreto uso las distros 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.

Instalación en Windows

Lo más recomendado es usar minGHC. Una vez instalado, podremos ejecutar GHCi en la consola de windows.

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> 2^128
340282366920938463463374607431768211456
```

Como vemos, Haskell soporta aritmética con enteros enormes (tanto como nuestra memoria principal nos permita). Como curiosidad, el número 2^128 (340282366920938463463374607431768211456) es el número de direcciones IP disponibles en el protocolo IPv6. A su vez se han ilustrado mediante ejemplos las funciones raíz cuadrada `sqrt` y la concatenación para cadenas `(++)`, que en este caso se escribe entre paréntesis al tratarse de un operador.

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

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

Como vemos, recibe dos argumentos, 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 comillas 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 argumentos 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
```

En Haskell, intentar usar `-13` como expresión nos dará un error, y debemos ponerlo entre paréntesis para que sea aceptado y evaluado convenientemente.

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.

Cambio de modo de GHCi

- `:set +s` cambia el modo de ejecución de GHCi, permitiendo ver cuánto tiempo tarda en ser evaluada cada una de nuestras expresiones. Muy útil para analizar el rendimiento de nuestro código.

Definiendo nuestras primeras funciones

Las funciones en Haskell se tratan como valores, como si de un `Bool`, `Char`, `Int`…se tratara. Lo único que las funciones pueden hacer en Haskell es trabajar con sus argumentos de entrada y devolver un valor de cierto tipo. Esto se acerca más al concepto de función matemática que por ejemplo las funciones en C/C++, que a parte de esto también tienen la capacidad de usar variables globales, escribir en pantalla…Esto no significa que en Haskell eso no se pueda hacer, pero hay que recurrir a soluciones específicas para que una función de Haskell sea capaz de llevar a cabo todas las mencionadas tareas.

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**:

```haskell
ochenta :: Int
```

Nota: tener en cuenta que Haskell es `case-sensitive`, es decir, sensible a mayúsculas y minúsculas. Los nombres de las funciones siempre deben empezar por minúscula, mientras que los módulos deben empezar siempre con mayúscula.

Las declaraciones de tipos suelen ser así:

```haskell
nombre_funcion :: parametroDeTipo1 -> parametroDeTipo2 -> ... -> parametroDeTipoN
```

Donde la función recibe un número N argumentos. De momento, vamos a pensar que el último argumento 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`:

```haskell
ochenta = 80
```

Como habíamos dicho, devuelve un valor de tipo `Int`, en este caso un 80 programado duramente (sin calcularlo, simplemente escribiendo un inmediato).

Nota: si vienes de lenguajes con sintaxis estilo C (C, C++…) podrías echar en falta la palabra reservada **return**. El `return` en Haskell es totalmente distinto, y ni siquiera implica que la función termine su ejecución.

Veamos ahora la definición completa que nos permitirá ver la función con mayor claridad.

```haskell
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. A parte de ello, la transparencia referencial nos permite conmutar aplicaciones de funciones por su definición, con la desventaja de necesitar paréntesis explícitos casi siempre.

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

```haskell
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.

Sistema de tipos

Haskell es un lenguaje fuertemente tipado, lo cual es su mayor ventaja de cara a conseguir una tasa de error muy alta en tiempo de compilación y baja en el tiempo de ejecución. La regla general es “si compila, suele funcionar”.

TODO.

Mencionar tipos y constantes polimórficas, y su uso.

```
fromIntegral :: (Integral a, Num b) => a -> b
```

Lambdas

En un lenguaje funcional, poner nombre a todas las funciones que usemos podría resultar tedioso. Además, es cómodo definir funciones al vuelo, es decir, rápidamente y en el punto del programa en el que realmente sea necesario. Las lambdas sirven para este propósito.

Las lambdas se suelen declarar entre paréntesis para que el compilador sepa que se tratan de un “todo”. No obstante, en el caso de las mónadas hay veces en que no son demasiado necesarios y se va resolviendo todo mediante la indentación.

La sintaxis de las lambdas es la siguiente:

```
(\arg1 arg2 ... argn -> cuerpo_función)
```

Es decir, para que Haskell sepa que estamos trabajando con una lambda, se usa la backslash `\` y a continuación se encuentran dos partes bien diferenciadas, separadas por una flecha `->`:

- A la izquierda de la flecha `->` la lista de nombres de argumentos, separados por espacios.
- A la derecha de la flecha `->` el cuerpo de la función. El tipo de esa expresión será el tipo retorno de la lambda.

Definamos la lambda más sencilla que existe, lo único que hace es devolver su argumento:

```
\x -> x
```

Definamos una lambda que eleve al cubo un número:

```
\x -> x*x*x
```

Veamos ahora una lambda que sume sus dos argumentos:

```
\x y -> x + y
```

Nota: el símbolo _ es el patrón subrayado, es decir, un patrón que casa con cualquier cosa y no guarda nada en memoria. Si desea saber más, lea la sección contigua “Reconocimiento de patrones”.

Y por último, veamos una que ignora su primer argumento y devuelva el segundo:

```
\_ x -> x
```

Como vemos, se puede usar el patrón subrayado para expresar que no nos importa el valor del primer argumento, ya que sólo usamos el segundo. Las lambdas tienen mucha importancia en Haskell, y se usarán bastante a lo largo de este tutorial.

La abstracción de una lambda “multi-argumento”

```
\x y z -> ...
```

es realmente azúcar sintáctico para

```
\x -> (\y -> (\z -> ...))
```

De igual modo, la definición de función

```
f x y z = ...
```

es azúcar sintáctico para

```
f = \x -> (\y -> (\z -> ...)).
```

Reconocimiento de patrones

Para entender qué es el reconocimiento de patrones primero debemos saber qué es *casar*. Para ello, el diccionario de la Real Academia es nuestro hamijo:

- Dicho de dos o más cosas: Corresponder, conformarse, cuadrar.
- Unir, juntar o hacer coincidir algo con otra cosa. Casar la oferta con la demanda.
- Disponer y ordenar algo de suerte que haga juego con otra cosa o tengan correspondencia entre sí.

Es un término que se usa bastante en las expresiones regulares, para ver si una expresión casa con un texto dado, y en qué lugar. Veamos un ejemplo de reconocimiento de patrones:

```
dime :: Int -> String
dime 1 = "¡Uno!"
dime 2 = "¡Dos!"
dime 3 = "¡Tres!"
dime 4 = "¡Cuatro!"
dime 5 = "¡Cinco!"
dime x = "No está entre 1 y 5"
```

La función `dime` hace reconocimiento de patrones con su primer argumento, de tipo `Int`, y va de arriba a abajo intentando encontrar una coincidencia. Cuando recibe un número entre 1 y 5, lo canta con ahínco, si no lo encuentra, nos devolverá un mensaje diciéndonoslo. Notar además que si hubiéramos puesto la línea `dime x = "No está entre 1 y 5"` al principio, nuestra función siempre devolvería `"No está entre 1 y 5"`, aun siendo cierto. Por tanto, debemos ordenar los patrones por probabilidad; de los menos probables a los más probables.

Cuando hablamos de reconocimiento de patrones hablamos, en realidad, de reconocimiento de constructores. En concreto en Haskell existen dos tipos de constructores, los constructores de tipos (los tipos que aparecen en las declaraciones de las funciones) y los constructores de valor (aquellos que se suelen poner entre paréntesis, y son funciones que recibiendo un valor crean un tipo que encapsula dicho valor). Deben empezar por mayúsculas, y en el caso de los constructores de valor se comportan como funciones de los tipos que contienen al tipo definido por el constructor de tipos. TODO.

Truco: realmente, debemos escribir tantas ecuaciones como constructores de valor en una función que reciba tipos algebraicos.

```
data Persona = CrearPersona String Int
--                          |      |
--                          |      |
--                          |      La edad de la persona
--                          El nombre de la persona
```

A la izquierda del igual está el constructor de tipos. A la derecha del igual están los constructores de datos. El constructor de tipos es el nombre del tipo y usado en las declaraciones de tipos. Los constructores de datos son funciones que producen valores del tipo dado. Si solo hay un constructor de datos, podemos llamarlo igual que el de tipo, ya que es imposible sustituirlos sintácticamente (recuerda, los constructores de tipos van en las declaraciones, los constructores de valor en las ecuaciones).

```
data Persona = Persona String Int
--   |         |
--   |         Constructor de datos
--   |
--   Constructor de tipos
```

El tipo del último ejemplo se conoce como **tipo de dato algebraico**; tipos de datos construidos mediante la combinación de otros tipos. El reconocimiento de patrones es una manera de desestructurar un tipo de dato algebraico, seleccionar una ecuación basada en su constructor y luego enlazar los componentes a variables. Cualquier constructor puede aparecer en un patrón; ese patrón casa con un valor si la etiqueta del patrón es la misma que la etiqueta del valor y todos los subpatrones casan con sus correspondientes componentes.

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

Variables de tipo

Las variables de tipo son aquellas que se declaran en `data` después del nombre del tipo que vamos a crear. Su finalidad principal es hacer saber qué puede formar parte del tipo, y además permitir a cualquier tipo formar parte de nuestro tipo personalizado. Veámoslo con un ejemplo:

```
data Persona a = PersonaConCosa String a | PersonaSinCosa String
--           |                         |
--           |                         podemos usarla aquí
--           |
--           Añadiendo una variable de tipo aquí
```

En los siguientes ejemplos se ilustra el deber de informar al compilador qué tipo queremos que nuestra función devuelva, y así producir un tipo `Persona Int`, `Persona String`,…,etc.

```
franConEdad :: Persona Int
franConEdad = PersonaConCosa "fran" 25

franSinEdad :: Persona Int
franSinEdad = PersonaSinCosa "fran"
```

Constructores de valor como funciones

Pues resulta que los constructores de valor son en realidad funciones, ¡qué sorpresa! En el ejemplo anterior, teníamos los constructores de valor `PersonaConCosa` y `PersonaSinCosa`. Veamos qué ocurre cuando inspeccionamos su tipo gracias al comando `:t` de GHCi:

```
*Main> :t PersonaConCosa
PersonaConCosa :: String -> a -> Persona a
*Main> :t PersonaSinCosa
PersonaSinCosa :: String -> Persona a
```

Como vemos, no hay demasiada diferencia con respecto a cualquier otra función, salvo su **letra inicial**, que debe ser ***mayúscula**. De este modo ya contamos con dos funciones , `PersonaConCosa` y `PersonaSinCosa`, que nos permiten construir un nuevo valor de tipo `Persona a` pasándoles, respectivamente `String -> a` y `String`.

Ahora llega el reconocimiento de patrones propiamente dicho; según se encuentre el constructor `PersonaConCosa String a` ó `PersonaSinCosa String`, nuestra función debe ser programada para actuar en consecuencia:

```
getNombre :: Persona Int -> String
getNombre (PersonaConCosa nombre _) = nombre
getNombre (PersonaSinCosa nombre)   = nombre

getEdad :: Persona Int -> Maybe Int
getEdad (PersonaConCosa _ edad) = Just edad
getEdad (PersonaSinCosa _)      = Nothing
```

Como vemos, a las variables `nombre` y `edad` respectivamente se le han *enlazado* sus valores reales, que son los que nuestra función devuelve. Como el constructor `PersonaSinCosa` sólo contiene el nombre y no la edad, utilizamos el tipo `Maybe` para devolver `Nothing` en caso de que ese patrón (constructor) sea reconocido. En el otro caso, devolvemos `Just edad` ya que en este caso la tenemos.

Importante: los paréntesis son obligatorios en este caso, pues se deben tratar como un todo (un valor del tipo `Persona a`). Aunque los patrones puedan sonarnos a llamada a función, **no lo son**, y el estar en la parte izquierda de la ecuación nos da pistas sobre ello.

Cabe destacar también que las funciones que reciban un argumento `Persona a` deben tener tantas ecuaciones de reconocimiento de patrones como constructores de valor haya en ese tipo de dato. No te preocupes demasiado por el número de patrones que tengas que reconocer, pues si el tipo está bien construido el número de ecuaciones será asumible.

En este caso su aridad es distinta, pues uno recibe dos argumentos y el otro uno, pero también podrían ser distintos si tuvieran el mismo número de argumentos pero estos fueran de distinto tipo.

Otro ejemplo de reconocimiento de patrones

Échale un ojo al siguiente código:

```
type Nombre = String

data Aniversario = Cumpleanios Nombre Fecha
                 | Boda Nombre Nombre Fecha

data Fecha = Fecha Int Int Int   -- Año, mes, día

pacoPerez :: Aniversario
pacoPerez = Cumpleanios "Paco Pérez" (Fecha 1968 7 3)

bodaPaco :: Aniversario
bodaPaco = Boda "Paco Pérez" "Juana López" (Fecha 1987 3 4)

type ListaAniversarios = [Aniversario]

anniversariesOfpacoPerez :: ListaAniversarios
anniversariesOfpacoPerez = [pacoPerez, bodaPaco]

mostrarFecha :: Fecha -> String
mostrarFecha (Fecha a m d) = show a ++ "-" ++ show m ++ "-" ++ show d

mostrarAniversario :: Aniversario -> String
mostrarAniversario (Cumpleanios nombre fecha) =
   nombre ++ " born " ++ mostrarFecha fecha
mostrarAniversario (Boda nombre1 nombre2 fecha) =
   nombre1 ++ " married " ++ nombre2 ++ " on " ++ mostrarFecha fecha
```

Escribiendo código por el estilo conseguimos código:

- más legible.
- cabeceras de funciones más descriptivas, casi autoexplicativas.
- que obliga al programador que trabaje con esos tipos a no alterar accidentalmente el orden de los argumentos.

Nota: vemos que en `Fecha` el constructor de tipo y de valor tienen el mismo nombre. Esto se considera una buena práctica solo cuando hay un solo constructor de valor para ese tipo.

Orden de ejecución del reconocimiento de patrones

Los patrones se pueden anidar con profundidad arbitraria, con el casamiento ejecutándose en el siguiente orden:

- dentro -> fuera
- izquierda -> derecha

Las ecuaciones de una definición de función se intentan en orden textual (de arriba a abajo), hasta que uno de los patrones case.

Guardianes vs reconocimiento de patrones vs expresiones case

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.

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.

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`.

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 argumento. 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 []
[]
```

Span

```
span :: (a -> Bool) -> [a] -> ([a], [a])
```

`span`, aplicada a un predicado `p` y a una lista `xs`, devuelve una tupla cuyo primer elemento es una lista con el mayor prefijo (posiblemente vacío) de `xs` de elementos que satisfacen `p` y su segundo elemento es el resto de la lista

```
span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
span (< 9) [1,2,3] == ([1,2,3],[])
span (< 0) [1,2,3] == ([],[1,2,3])
```

`span p xs` es equivalente a `(takeWhile p xs, dropWhile p xs)`

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. En el último caso se suele poner la palabra reservada `otherwise`, que es simplemente un alias de `True`. De este modo, será un caso que siempre se ejecutará en el caso de que no se cumpla ninguna de las otras condiciones.

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 argumento 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
```

Composición de funciones

La función composición, `(.)`, crea una función que recibe como argumentos dos funciones, `f` y `g` y devuelve una **nueva función**, `h`, que se puede notar `h = f . g`.

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

Veamos la definición de la función `(.)` para aclarar conceptos:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

Veamos un sencillo ejemplo aclaratorio:

```
f :: Int -> Int
f = (+13)

g :: Int -> Int
g = (*7)

h :: Int -> Int
h = f . g
```

Veamos un ejemplo en acción:

```
*Main> h 10
83
```

TODO. El libro da unas restricciones de argumentos y valores de retorno muy buenas.

Importante: la composición es asociativa a la derecha.

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, 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 argumentos:

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 argumentos. Delante de ella escribimos `$` y componemos todas las funciones que venían antes escribiéndolas sin su último argumento 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 `(.)`.

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.

Ahora pasamos a ver dos versiones de la función `quitarDuplicados`, 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
```

Esta versión de `quitarDuplicados` tiene la ventaja de ser muy concisa y casi autoexplicativa.

Funciones de orden superior

Haskell se llama así por un matemático y lógico estadounidense llamado Haskell Brooks Curry, que inventó la técnica conocida como currificación.

Currificar equivale a fijar argumentos, dando lugar a nuevas funciones.

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

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

Importante: 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 argumento, nos devuelve otra función que recibe otro argumento de tipo `a`, y al hacer su cálculo devuelve otro también del tipo `a`.

Pereza

Se dice que la pereza es el mejor de los 7 pecados capitales, pues no nos permite cometer los otros 6. 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.
```

Nota: vemos que la salida contiene `Interrupted`, eso es fruto de pulsar la combinación de teclas (Ctrl + c) en sistemas POSIX, esta combinación interrumpe la ejecución del proceso en ejecución en una terminal.

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.

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 (sería divisible). Estos números irán a parar a la lista solución como `ks`, que serán los divisores, por tanto, desde que esa lista contenga un sólo elemento, ya el número no será primo. Haskell es *perezoso*, y nos aprovecharemos de ello; veamos la función `null` definida en el módulo `Prelude`:

```
null :: [a] -> Bool
null []    =  True
null (_:_) =  False
```

La función `null` recibe una lista, devolviendo `True` si se reconoce el patrón lista vacía (`[]`) y `False` en caso de que contenga al menos un elemento. Desde que la comprensión de listas tenga algún resultado, `null` entrará en acción, pues `(_:_)` casa con listas de un sólo elemento, donde el primer patrón subrayado casaría con la cabeza y el segundo patrón subrayado casaría con la lista vacía `[]`. Si esto ocurre, `null` devolverá `False` directamente, sin generar más `k` de esa lista.

En conclusión, hemos usado la cualidad de *perezoso* para conseguir una condición de parada que optimice nuestra función `esPrimo`.

El tipo Maybe

La función `find` devuelve el primer elemento que cumple un predicado que le pasamos como primer argumento, 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
*Main> 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]
```

Uso de librerías (biblotecas) externas

Las bibliotecas son conjuntos de funciones que fueron definidas por otras personas que han decidido compartir su código. En muchas ocasiones se trata de código muy probado y sencillo de usar, y el que veremos a continuación además nos da muy buen rendimiento.

Para instalar las últimas versiones de las librerías que utilicemos, usaremos el comando `cabal`, el cual es un gestor de paquetes para Haskell.

Lo primero que tenemos que hacer es mandar a `cabal` a actualizarse, lo cual se hace con el comando:

```
cabal update
```

Esto hará que se descargue la lista de paquetes más actuales desde hackage.

A continuación, procederemos a instalar paquetes. `cabal` descarga los ficheros fuente, los compila y configura. Instalaremos el paquete `primes` que tiene utilidades muy eficientes para trabajar con números primos:

```
cabal install primes
```

Para usar este paquete, deberemos añadir en la cabecera de nuestro programa la sentencia:

```
import Data.Numbers.Primes
```

Una vez hecho esto, podemos usar todas éstas funciones como si las tuviéramos definidas.

Para ver todos los paquetes disponibles, podemos consultar la página oficial.

`Data.Numbers.Primes` para resolver el problema 10 de Project Euler, a continuación el programa completo:

```haskell
import Data.List (takeWhile)
import Data.Numbers.Primes (primes)

sumaPrimos = sum (takeWhile (< 2000000) primes)
```

Como se aprecia, después de los `import` de los módulos, entre paréntesis especifico qué función quiero (en este caso sólo una, si hubiera más, irían separadas por comas). De este modo, en vez de cargar el módulo completo, cargo sólo lo que me interesa.

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)
```

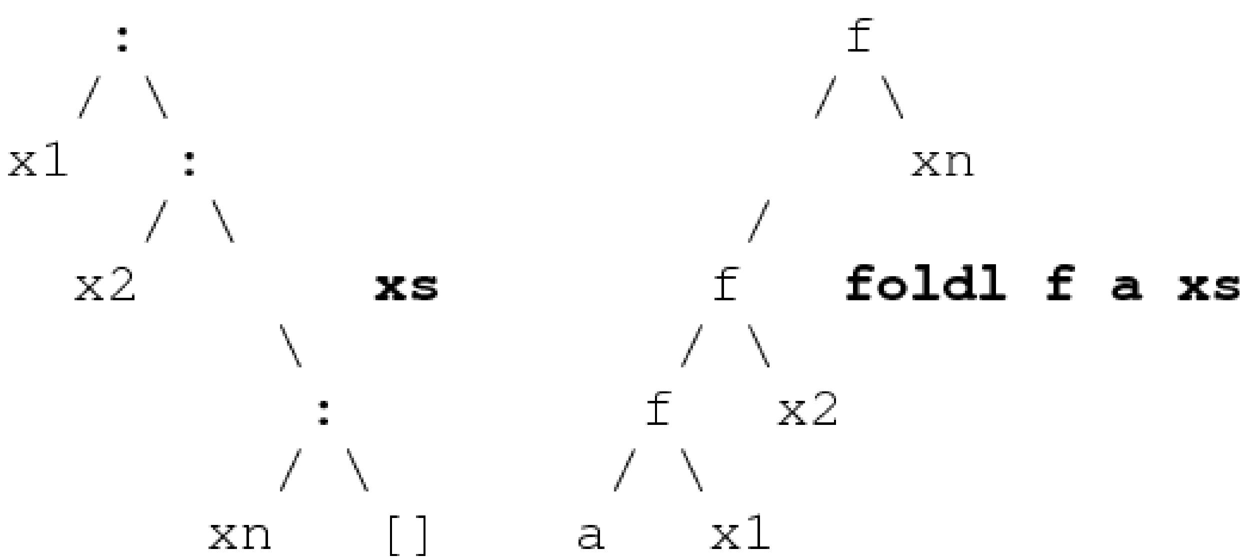

Figure 1: Despliegue de foldl

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

```haskell
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:

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

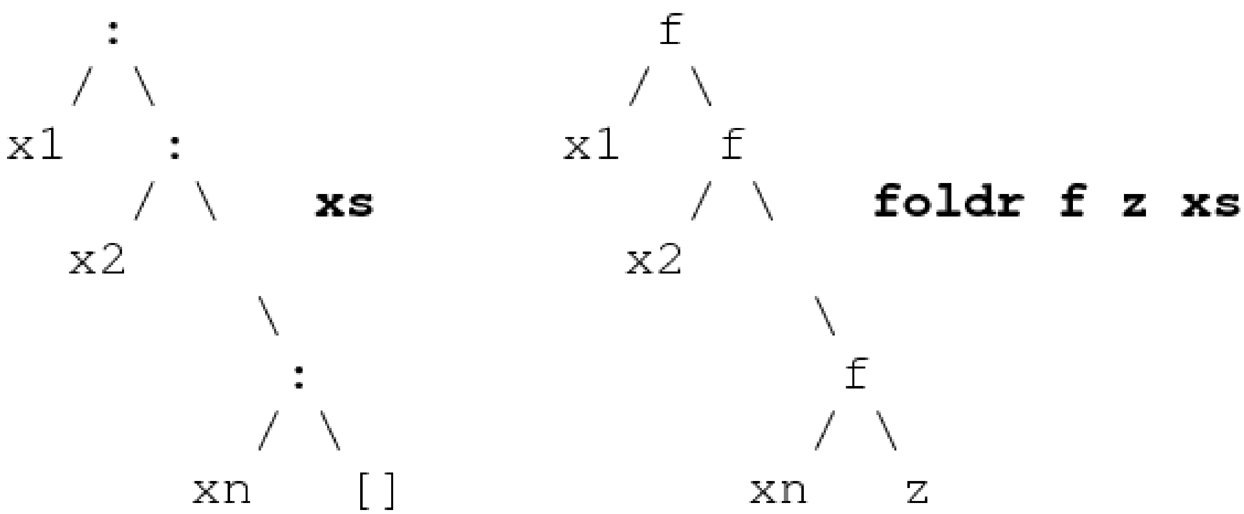

Figure 2: Despliegue de foldr

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 argumento **derecho** (segundo argumento de la función binaria), y en el **foldl** (plegado a la **izquierda**) el acumulador es el argumento **izquierdo** (primer argumento 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 currificicación.

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 argumento para dar un resultado fijo. Por ejemplo, a `&&` no le importa el valor de su segundo argumento si el valor del primer argumento 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 argumento. 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 argumento (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 argumentos 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 argumentos de tipo.

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

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

Decimos que un tipo es concreto si no recibe argumentos de tipo (como `Char` o `Bool`), o si recibe argumentos 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…), sino con las interfaces de Java.

Clase Ord

En una declaración como esta:

```haskell
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 argumentos de tipo, en caso de que los tuvieran, se compararían estos argumentos (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 =
  [("Ana", "661898252")
  ,("María", "637562754")
  ,("Níove", "686457823")
  ,("Rut", "696443312")
  ,("Miriam", "822063519")
  ,("Juana", "626789321")
  ]
```

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 su significado viene a ser el primero usado como pronombre, es decir, o es uno u otro. `Either` 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`. En `Either`, se suele usar el constructor de valor `Left` para denotar error, mientras que `Right` indica computación exitosa.

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 argumento.

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 argumentos 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.

A consecuencia de que Haskell goza de transparencia referencial, podemos reemplazar funciones con sus implementaciones de manera gratuita. Quizá requiera paréntesis explícitos aquí y allá, pero el código siempre dará la misma salida.

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.

```haskell
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

```haskell
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"
```

```haskell
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`.

```haskell
getStdGen :: IO StdGen
```

Veamos cómo quedaría el programa:

```haskell
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.

```haskell
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.

```haskell
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).

```haskell
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 byte (8 bits). Además, el modo en que son vagas también es distinto.

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**.

Hay dos modos fundamentales de ver `fmap`, `fmap` es una función que:

- recibe dos argumentos, una función y un contenedor, y aplica la función "dentro" del contenedor, sobre todos sus elementos, para devolver un nuevo contenedor.
- está currificada (como todas las demás), así que realmente no recibe dos argumentos, sino que simplemente recibe una función de tipo `a -> b` para devolver otra función de tipo `f a -> f b` donde `f` es la instancia de la clase de tipos `Functor` sobre la cual estamos trabajando. Quizás esto no sea tan obvio, pero se puede llegar a esta conclusión parentizando la cabecera de `fmap` de este modo: `fmap :: (a -> b) -> (f a -> f b)`.

Nota: el segundo modo de ver `fmap` es conocido como "lifting", es decir, elevamos una función "normal" `g :: a -> b` a otra función de tipo `fmap g :: f a -> f b`.

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 argumento. Por tanto una función `a -> b -> c` recibe un único argumento de tipo a y devuelve una función `b -> c`, que coge un argumento y devuelve `c`. Por tanto, la aplicación parcial de funciones devuelve una función que toma los argumentos 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 argumento 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`:

```haskell
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 argumentos 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 “multiargumento” sobre valores funtor, obtenemos valores funtor que contienen funciones.

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

```
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

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

Vemos que `f` juega el rol del funtor aplicativo, y debe recibir un tipo concreto como argumento, 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 currificación (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 currificación, si a la función `+` sólo le aplicamos un `3` va a darnos una función que recibe un argumento y le suma 3 a dicho argumento.

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 argumentos no aplicativos y usar esa función para operar en muchos valores aplicativos. La función puede recibir tantos argumentos 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:

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

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 argumentos 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 argumento 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 comprensión de listas. `<*>` debe, de alguna manera, extraer la función del argumento izquierdo y luego mapearla sobre el argumento derecho. En este caso, como el argumento 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 argumento derecho completa. Esto se repite hasta que la lista argumento 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 argumentos, 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 argumento 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 argumento `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 argumento izquierdo y aplicarla sobre su argumento 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 currificación, 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 argumento 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):

```haskell
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 pasemos 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 argumentos y "zipea" dos listas con ella. `zipWith3` recibe una función con tres argumentos 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 funtores aplicativos

`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:

```haskell
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:

```haskell
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 comprensión 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.

Otros operadores sobre funtores aplicativos

Truco: si hay un mayor que '>' o menor que '<', el resultado al lado al que apunte este signo deberá ser usado. Por ejemplo, `*>` devuelve el resultado a su derecha; `<*>` devuelve los resultados de ambos lados; y `<*` devuelve el resultado a su izquierda. Además, el uso de estos operadores no cambia el orden en que las acciones a la izquierda y derecha de los mismos, que siempre será de izquierda a derecha. Por ejemplo; `(parser <* spaces)` ejecutará `parser` primero y luego `spaces`, quedándose con el resultado de `parser`.

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 argumentos 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 argumento 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 currificación; 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 argumentos 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 currificados (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). Lo único que necesita un lenguaje de programación para soportar mónadas es la capacidad de usar lambdas.

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 argumento 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 la definición de cada método, sustituyendo el argumento `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`.

```haskell
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):

```haskell
(>>=) :: 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

`haskell (>>) :: m a -> m b -> m b m >> k = m >>= \_ -> k`haskell

Así que es sólo para ahorrarse 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`.

`(>>)` **no** es simplemente ignorar el primer argumento. Lo hacemos por definición:

```haskell
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 argumento y devuelve `Id 4` como resultado. Es importante remarcar que **no todas las mónadas** ignoran su primer argumento 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`, en un contexto monádico, gracias a `return`.

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:

`haskell Just :: a -> Maybe a Nothing :: Maybe a`haskell

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`:

```haskell
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 hubiera 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 exite, 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:

```haskell
return :: a -> m a
```

Por tanto podríamos definir:

```haskell
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:

```haskell
(>>=) :: M a -> (a -> M b) -> M b
```

Por tanto, si pasamos eso a nuestra futura mónada `Maybe`:

```haskell
(>>=) :: 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 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:

```haskell
(>>) :: 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.

```haskell
(>>) :: Maybe a -> Maybe b -> Maybe b
m >> k = m >>= \_ -> k
```

Vayamos un poco más allá y veamos qué pasa si sustituímos el operador bind:

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing (>>=) _  = Nothing
(Just x) (>>=) f = f x
```

Luego, de `m >> k = m >>= \_ -> k` nos quedamos con:

```haskell
m >>= \_ -> k
```

Que se sustituye por:

```haskell
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:

```haskell
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?

`haskell do c1 <- gooddiv a1 b1 c2 <- gooddiv a2 b2 return (c1 + c2)`haskell

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:

```haskell
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`:

```haskell
(>>=) :: 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 función mapM

`mapM` mapea una función monádica sobre una lista de valores, secuencia las acciones resultantes a través de `(>>=)`, y luego devuelve una mónada que contiene la lista de los resultados internos.

```
mapM             :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as        =  sequence (map f as)
```

Donde `sequence`:

```
    sequence       :: Monad m => [m a] -> m [a]
    sequence       =  foldr mcons (return [])
                        where mcons p q = p >>= \x -> q >>= \y -> return (x:y)
```

La función mapM_

```
    mapM_            :: Monad m => (a -> m b) -> [a] -> m ()
    mapM_ f as       =  sequence_ (map f as)
```

donde `sequence_`:

```
    sequence_      :: Monad m => [m a] -> m ()
    sequence_      =  foldr (>>) (return ())
```

La función (=<<)

Es en realidad `(>>=)` pero con el orden de los argumentos invertido:

```
    (=<<)            :: Monad m => (a -> m b) -> m a -> m b
    f =<< x          =  x >>= f
```

La mónada lista

Puede que te lo creas, puede que no, pero te digo que lo que vas a ver a continuación te saldrá de manera natural si programas algo relativamente complejo, como un NFA. De hecho las mónadas no son más que la generalización de un tipo dado de computación.

```
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 comprensión 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
```

Asociatividad de los elementos de Haskell

Asocian **a izquierdas:**

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

Asocian **a derechas:**

- En las declaraciones de tipos, `(->)`.
- 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.

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 argumentos formales de las funciones… Cuando un argumento formal se asocia con un valor de argumento 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 argumento 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 argumentos formales los cuales estarán "ligados" a nuevos valores reales de argumentos.

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 argumentos 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 argumentos 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 argumentos 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.

Evaluación perezosa: modo de funcionamiento

Si sabes programar en algún lenguaje imperativo es probable que sepas usar el “short circuit”, es decir, un método que permita reducir la evaluación de varias expresiones a algunas menos, e incluso en ciertas ocasiones a sólo una.

La evaluación perezosa trata de emular este efecto, pero no se limita a la evaluación de expresiones booleanas, sino a prácticamente todo el conjunto del lenguaje Haskell. Se pueden realizar computaciones con gran complejidad gracias a la evaluación perezosa.

En la evaluación perezosa las expresiones se evalúan de izquierda a derecha, construyendo un grafo. Luego, este grafo se empieza a reducir de arriba a abajo, y mediante esto se consigue muchas veces que ciertos nodos del árbol queden sin explorar, ya que no es necesario hacerlo, sólo con una parte del árbol podemos saber, en ciertos casos, qué resultado dará la expresión en su conjunto.

Teorema: La evaluación perezosa **nunca** ejecuta más pasos de evaluación que la evaluación impaciente.

Además de ahorrar pasos de evaluación, la evaluación perezosa nos permite trabajar con listas infinitas, algo que la evaluación impaciente no puede hacer.

Otra idea importante es que el compilador no traduce directamente el Haskell que programamos a código máquina, sino que lo cambia. Por ello es muy difícil saber qué hará realmente nuestro programa a bajo nivel.

La ejecución de un progrma en Haskell se basa en la evaluación de expresiones. La idea principal que rige la evaluación es la aplicación de funciones. Dada una función:

```
cuadrado x = x*x
```

Podemos evaluar la expresión:

```
cuadrado (1+2)
```

Reemplazando la parte izquierda de la definición de `cuadrado` con la de la derecha, y sustituyendo el argumento `x` por su argumento real.

```
cuadrado (1+2)
=> (1+2)*(1+2)
```

La evaluación ahora necesita hacer primero las sumas, aplicando `(+)`, para luego pasarle los resultados a `(*)`:

```
(1+2)*(1+2)
=> 3*(1+2)
=> 3*3
=> 9
```

En este caso, hemos evaluado `(1+2)` dos veces. Sin embargo, sabemos que las dos expresiones `(1+2)` son realmente la misma, porque corresponden al mismo argumento de la función, `x`.

Para evitar estos cálculos redudantes se usa un método llamado reducción de grafo. Cada expresión puede ser representada como un grafo. Nuestro ejemplo se puede representar como sigue.

Ideas sueltas

No usar head, init, last etc., ver el capítulo 2 de CIS 194 para reemplazar estas malas costumbres.

En general, la siguiente gramática define qué puede ser usado como patrón:

```
pat ::= _
     |  var
     |  var @ ( pat )
     |  ( Constructor pat1 pat2 ... patn )
```

Importante: mencionar el ejemplo del DFA en el cual tenemos que usar `let` dentro de un bloque `do` para que se tome lo que hay dentro como funciones efectivas.

`(<*>)` es el ya conocido `($)` elevado a funtores aplicativos, y `ap` es lo mismo pero elevado a mónadas.

```
ghci> :type ($)
($) :: (a -> b) -> a -> b
ghci> :type (<*>)
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
ghci> :type ap
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

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

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`).

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

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 argumentos de funciones usando el estilo aplicativo.

Las mónadas son superiores y nos permiten encadenar computaciones que podrían fallar, entre otras cosas, 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 comprensión 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]

En realidad las expresiones `case` son reconocimiento de patrones.

Ideas Razonando con Haskell

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
```

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 7:

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

Parsec

Parsec es un módulo de Haskell, un conjunto de funciones exportables que suelen tener una finalidad común y se pueden importar en otros programas.

Parsec se diseñó desde cero como una librería de parsers con capacidades industriales. Es simple, segura, está bien documentada, provee de buenos mensajes de error y es rápida. Se define como un transformador de mónadas que puede ser apilado sobre mónadas arbitrarias, y también es paramétrico en el tipo de flujo de entrada. La documentación de la versión usada en el presente Trabajo Fin de Grado se puede consultar online en hackage.

Parsec se puede leer en "inglés plano" (siempre que nuestros parsers tengan los nombres adecuados). Se pueden hacer analogías entre las funciones de Parsec y las expresiones regulares, como veremos en el ejemplo de código de este capítulo.

La mónada sobre la que opera Parsec es `GenParser`.

el operador `<-` (que es en realidad bind (>>=)) liga a un nombre lo que hay dentro de la mónada sobre la cual opera (GenParser). Por tanto, si opera sobre:

- `many`, `many1`, `string`…ligará una `String`
- `char`…ligará un `Char`
- `noneOf` NO consume aquella entrada que no debe, dicha entrada es una condición de parada.

`<|>` es el operador de elección. Pueden ser encadenados tantos parsers como queramos. Este operador lo que hace es:

1. intenta el parser de la izquierda, que no debería consumir entrada…(ver `try`)
2. intenta el parser de la derecha.

Si el parser de la izquierda consume entrada, podríamos usar `try`…intenta ejecutar ese Parser, y, si falla, vuelve al estado anterior, es decir, deja la entrada sin consumir. Sólo funciona a la izquierda de `<|>`, es decir, si queremos encadenar varios `try`, deben estar a la izquierda de la cadena de `<|>`.

`try` es como un lookahead, y se puede ver como algo para procesar cosas de manera atómica, `try` es realmente backtracking, y por ello no es demasiado eficiente.

`char`, `string`…consumen entrada, si pueden.

`(>>)` lo que hace es encadenar parsers, si tienen éxito, se ejecuta el siguiente. El parser que preceda a `>>` no ligará su resultado a ningún nombre. `>>` consume entrada, y falla si ambos argumentos (parsers) fallan.

```
type CharParser st a = GenParser Char st a
```

`<?>` lo que hace es informar de un error si es que se produce (en tiempo de ejecución).

`<$>` es sinónimo de `fmap`.

`between parser a c` el carácter de apertura, luego el parser, después el de cierre y se queda con lo que parser haya parseado.

`type ReadS a = String -> [(a, String)]`. En realidad esta función `reads` se trata de un parser, como su propio tipo indica.

Parsec en acción: un parser de JSON

Librerías necesarias

```haskell
import Text.ParserCombinators.Parsec hiding ((<|>), many)
import Text.Parsec.Numbers (parseFloat)
import Control.Applicative
import Control.Monad
import Prelude hiding (Null,null)
```

Formato a parsear

El formato JSON (JavaScript Object Notation) es de los más comunes hoy en día para el intercambio de información a través de la red. Es un formato sencillo y fácil de parsear, y por ello está ganando terreno frente a su competidor principal, XML. Sus principales elementos son:

- Number
- String
- Boolean
- Array
- Object
- null

La principal aplicación de Haskell siempre han sido los parsers, tanto para compiladores como para propósito general.

Empezaremos definiendo los parsers más sencillos, cuyo fin es devolver argumentos que entrarán en constructores de valor para tipos de JSON que siempre valgan lo mismo. Estos 3 tipos son: `true`, `false` y `null`.

```haskell
alwaysTrue :: Parser Bool
alwaysTrue = pure True

alwaysFalse :: Parser Bool
alwaysFalse = pure False

alwaysNull :: Parser String
alwaysNull = pure "null"
```

La misión de `pure :: a -> f a` (donde en este caso `f` es la mónada `Parser`) no es otra que envolver los dos `Bool` y la `String` en un valor monádico, devolviendo de este modo un `Parser Bool` o un `Parser String`. Por tanto, estas funciones devuelven un `Parser`, que cuando se ejecuta (mediante la función parse) devuelve un `Bool` o una `String`.

Ahora lo que debemos hacer es usar el parser `string`, que intenta casar con una cadena dada, devolviéndola en caso de que consiga casar:

```
matchTrue :: Parser String
matchTrue = string "true"

matchFalse :: Parser String
matchFalse = string "false"

matchNull :: Parser String
matchNull = string "null"
```

Por último, no devolveremos la cadena propiamente dicha, sino un valor puro (por ello antes definimos funciones que usan `pure`):

```
boolTrue :: Parser Bool
boolTrue = matchTrue *> alwaysTrue

boolFalse :: Parser Bool
boolFalse = matchFalse *> alwaysFalse

null :: Parser String
null = matchNull *> alwaysNull
```

Aquí usamos un operador de la clase de tipos `Applicative`, que en inglés se suele llamar "star arrow". Este operador ejecuta primero el parser de la izquierda, luego el de la derecha, y devuelve sólo lo que parsee el de la derecha (el sitio al que apunta la flecha).

Ahora veamos qué pasa si un token puede pertenecer a un tipo aún más general:

```
bool :: Parser Bool
bool = boolTrue <|> boolFalse
```

`<|>` es el operador de elección, y se parece mucho a la barra vertical `|` de las expresiones regulares. Pueden ser encadenados tantos parsers como queramos. Este operador lo que hace es:

1. intenta el parser de la izquierda, que no debería consumir entrada…(ver `try`)
2. intenta el parser de la derecha.

Si el parser de la izquierda consume entrada, podríamos usar `try`…intenta ejecutar ese Parser, y, si falla, vuelve al estado anterior, es decir, deja la entrada sin consumir. Sólo funciona a la izquierda de `<|>`, es decir, si queremos encadenar varios `try`, deben estar a la izquierda de la cadena de `<|>`.

`try` es como un lookahead, y se puede ver como algo para procesar cosas de manera atómica, `try` es realmente backtracking, y por ello no es demasiado eficiente.

Como en este caso las string que vamos a parsear no tienen prefijos coincidentes, no hace falta usar `try` por si hay que volver a empezar.

Ahora empezaremos a ver algo que se parece aún más a las expresiones regulares:

```haskell
stringLiteral :: Parser String
stringLiteral = char '"' *> (many (noneOf ['"'])) <* char '"'
```

Aquí vemos que Parsec puede leerse casi en "inglés plano", ya que esta línea casi se autodescribe. Primero, debe encontrarse un carácter ", luego vemos la función `many`, que equivale a la estrella `*` de las expresiones regulares, es decir, podría haber muchos, uno o ninguna ocurrencia del parser que reciba `many`.

Luego vemos una función `noneOf`, que es un parser que acepta todo menos los carácteres que pertenezcan a una lista determinada, en este caso acepta todo menos las comillas dobles, en caso de toparse con comillas dobles (la cadena ha acabado), deja de consumir entrada.

Para terminar, se vuelve a parsear un carácter ", que debe estar obligatoriamente. Ahora vemos que nuestra combinación aplicativa sigue una estructura `a >* b <* c`, esto indica que los parsers `a`, `b` y `c` deben tener éxito, pero como sólo se devuelve lo que está apuntado por las flechas, sólo devolveremos lo que haya parseado `b`, que en este caso corresponde a `(many (noneOf ['"']))`.

De modo que Parsec, como la programación funcional, se basa en hacer sencillas funciones que sean buenas en lo suyo, e irlas combinando para realizar tareas cada vez más complejas. Esta es la base de la filosofía KISS tan popular en los sistemas POSIX.

La siguiente línea da error de tipos:

```haskell
value = bool <|> stringLiteral
```

Solución: crear un tipo algebraico que contenga Bool y String (entre otros). Los tipos de datos algebraicos son una herramienta muy útil para los parsers, ya que permiten saber exactamente a qué tipo pertenece el token que hemos parseado.

```haskell
data JSONValue = B Bool
               | S String
               | N Double --número de JSON
               | A [JSONValue] --array de JSON
               | O [(String, JSONValue)] --objeto de JSON
               | Null String
               deriving Show
```

Como vemos, tenemos un sólo constructor de tipo, `JSONValue`, es decir, nuestros parsers tendrán tipo `Parser JSONValue`. Sin embargo, tenemos 6 constructores de valor, que por simplicidad son simplemente las letras Iniciales de cada tipo de valor a parsear, salvo `Null`, en el cual se usó el nombre completo ya que `N` se usó para el tipo Number de JavaScript.

Veamos ahora el parser principal, es decir, un parser genérico capaz de parsear cualquier valor de JSON:

```haskell
jsonValue :: Parser JSONValue
jsonValue = spaces >> (jsonNull
                   <|> jsonBool
                   <|> jsonStringLiteral
                   <|> jsonArray
                   <|> jsonObject
                   <|> jsonNumber
                   <?> "JSON value")
```

No te preocupes demasiado por el parser `spaces`, lo explicaré más adelante en conjunto con `lexeme`. Pero, ¿qué es esa interrogación? `<?>` es un combinador que permite dar mensajes de error en caso de parseo fallido. En este caso, se le pasa una `String` con el mensaje de error que queremos que aparezca. En caso de error, saldrá algo como "expected JSON value", pues ese es el argumento de `<?>` para cuando falle el parser `jsonValue`.

Lo malo de esto es que seguimos teniendo error de tipos porque:

```
bool :: Parser Bool
stringLiteral :: Parser String
```

Lo bueno es que con `(<$>) :: Functor f => (a -> b) -> f a -> f b`, que en este caso tendría el tipo: `(<$>) :: (a -> b) -> Parser a -> Parser b`, podemos solucionarlo.

Recordemos que los constructores de valor son en realidad funciones como otra cualquiera (salvo que empiezan por mayúscula). Por ejemplo, el tipo de `B` es `B :: Bool -> JSONValue`. Por tanto, si hacemos `B <$> bool` tendremos como resultado un `Parser JSONValue`, y eso haremos en todos nuestros parsers anteriormente nombrados.

```
jsonBool'' :: Parser JSONValue
jsonBool'' = B <$> bool

matchNull'' = lexeme matchNull'

jsonStringLiteral :: Parser JSONValue
jsonStringLiteral = lexeme (S <$> stringLiteral)
```

Aquí lo único que nos llama la atención es el parser `lexeme`. `lexeme` está definido en Parsec por defecto, pero nosotros lo programaremos más que nada por razones didácticas.

`lexeme` es un parser que, recibiendo otro parser, devuelve un parser del mismo tipo, pero que consume todos los espacios (incluyendo tabuladores y newlines) que haya detrás del token parseado.

```
ws :: Parser String -- whitespace
ws = many (oneOf " \t\n")

lexeme :: Parser a -> Parser a
lexeme p = p <* ws
```

De este modo, con aplicar lexeme a cada uno de los parsers que vayamos a usar, tenemos resuelto el problema de los espacios entre tokens.

Bueno, ahora que el problema de los espacios está resuelto…¡sorpresa! no lo está del todo…Como hemos dicho, el combinador `lexeme` se "come" todos los espacios, tabuladores o newlines que encuentre después del token parseado. Pero, ¿y si esos espacios estuvieran antes del primer token que llegamos a parsear? Probablemente se produciría un error.

Solución: añadir el parser `spaces` a nuestro parser principal `jsonValue`. Esto se hizo mediante el operador monádico `>>`, que en la mónada de Parsec tiene el efecto de ejecutar ese parser, y si tiene éxito no guardar el resultado del parsing, sino pasar al siguiente. Se ha usado `>>` para ilustrar el uso de esta función, ya que se había introducido antes `*>`.

A continuación creemos un parser que permita parsear números. Para ello usaremos la función `parseFloat`, que permite parsear cualquier tipo de número, incluso con signo, exponente, parte decimal…es decir, el formato de coma flotante.

```
jsonNumber :: Parser JSONValue
jsonNumber = N <$> parseFloat
```

¡Listo! ya tenemos un parser más. Ahora veamos algo un poco más complejo, los arrays de JSON. Un array de JSON tiene el siguiente formato:

```
[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter","lastName":"Jones"}
]
```

Como vemos, tenemos:

1. Un carácter abrir corchetes `[`
2. Un conjunto de tokens de JSON, separados por comas.
3. Un carácter cerrar corchetes `]`

Sabido esto, lo único nuevo que tenemos que introducir aquí es el parser `sepBy`. `sepBy` recibe dos argumentos, el primero es el parser que se usará para cada token, y el segundo es el parser que se usará para el separador o separadores. Veamos el parser completo.

```haskell
array :: Parser [JSONValue]
array =
  (lexeme $ char '[')
  *>
  ( jsonValue `sepBy` (lexeme $ char ',') )
  <*
  (lexeme $ char ']')
```

Como los arrays contienen tokens de JSON, lo que hacemos es una llamada recursiva a `jsonValue`. De este modo, vemos que dentro de un array de JSON puede haber "lo que sea" (siempre que esté correctamente escrito y formateado) pero el array debe empezar por el carácter `[` y terminar con `]` para garantizar que dicho formato sea correcto. Como vemos, este parser nos devuelve una lista de `JSONValue`, y eso no es un tipo `JSONValue`. Por tanto, debemos aplicar `fmap`, en este caso de manera infija:

```haskell
jsonArray :: Parser JSONValue
jsonArray = A <$> array
```

Ahora parsearemos algo parecido pero no del todo igual, los objetos de JSON. El formato de los objetos es:

1. Un carácter abrir llaves `{`
2. Una lista de pares separador por el carácter dos puntos ':'
3. Un carácter cerrar llaves `}`

```haskell
jsonObject :: Parser JSONValue
jsonObject = O <$> ((lexeme $ char '{') *>
                    (objectEntry `sepBy` (lexeme $ char ','))
                    <* (lexeme $ char '}'))

objectEntry :: Parser (String, JSONValue)
objectEntry = do
  key <- lexeme stringLiteral -- >>=
  char ':'                    -- >>
  value <- lexeme jsonValue
  return (key, value)
```

Ahora consigamos que el parser `jsonBool'` sea capaz de lidiar con espacios, tabuladores y nuevas líneas después del token que parsea:

```
jsonBool' = lexeme jsonBool''
```

Ya casi hemos terminado, pero aún falta un pequeño detalle. ¿Y si alguien se equivoca y escribe por ejemplo "falseee", o "nullpointer", o cualquier otra cosa siguiendo a las palabras reservadas `true`, `false` o `null`? Nuestro parser lo aceptaría, cuando eso no debería ser así. Queremos exactamente esas palabras, ni un carácter más ni uno menos, para que nuestro parser sea correcto. Para ello, Parsec nos provee con un parser que falla en caso de que otro esté seguido de ciertos carácteres, es `notFollowedBy`. `notFollowedBy` recibe un parser, y si éste tiene éxito, falla. Un ingenioso truco que nos saca del atolladero de manera muy sencilla y casi autoexplicativa.

```
jsonBool :: Parser JSONValue
jsonBool = jsonBool' <* notFollowedBy alphaNum

jsonNull :: Parser JSONValue
jsonNull = matchNull'' <* notFollowedBy alphaNum
```

Por último, creemos la función `main` que nos permitirá compilar el programa. Para ello seguiremos los siguientes pasos:

1. Mostrar por pantalla qué queremos.
2. Obtener el nombre del fichero por entrada estándar (teclado) y ligarlo al nombre `filename`.
3. Aplicar nuestro parser principal (`jsonValue`) a nuestro fichero `filename` mediante la función `parseFromFile`.

```
main = do
  putStr "Nombre_fichero: "
  filename <- getLine
  parseFromFile jsonValue filename
```

Y listo, ya tenemos nuestro parser de JSON funcionando.

Felicidades por haber completado el tutorial (o por haber saltado a la última página, que nos conocemos xD) y espero que haya sido de tu agrado. Si tienes cualquier duda o sugerencia, estoy disponible en freinn@gmail.com.