¡Aprende Haskell por el bien de todos!

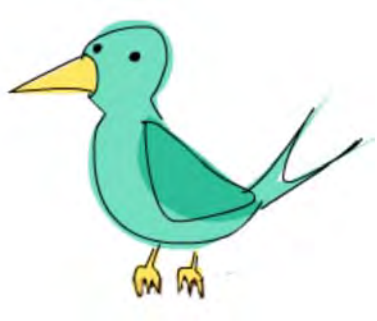
¡Aprende Haskell por el bien de todos!

Introducción

Sobre esta guía

Bienvenido a *¡Aprende Haskell por el bien de todos!* Si estás leyendo esto probablemente quieras aprender Haskell. Pues bien, has venido al sitio adecuado, pero primero vamos a hablar un poco sobre esta guía.

Decidí escribir esta guía porque quería arraigar mi propio conocimiento de Haskell y porque pensé que podía ayudar a la gente que empezaba con Haskell. Existen bastantes manuales y guías sobre Haskell por la red. Cuando empecé con Haskell no lo leí un único documento. La forma en la que aprendí Haskell fue leyendo varios artículos y guías, porque explicaban el mismo concepto de diferentes formas. Así, yendo a través de varios documentos, fui capaz de juntar todas las piezas y entonces todo encajó. De modo que, esto es un intento más de añadir otro útil documento para aprender Haskell de forma que tengas más oportunidades de encontrar uno que te guste.



Esta guía está dirigida a personas que tengan experiencia en lenguajes de programación imperativa (C, C++, Java, Python...) pero que no hayan programado antes en ningún lenguaje funcional (Haskell, ML, OCaml...). Aunque apuesto que incluso si no tienes experiencia como programador, un tipo inteligente como tú podrá seguir adelante y aprender Haskell.

El canal *#Haskell* de la red *freenode* es un buen lugar para preguntar dudas si te sientes estancado y sabes inglés. La gente es bastante amable, paciente y comprensible con los que empiezan.

Intenté aprender Haskell, dos veces, antes de conseguir entenderlo. Al principio todo parecía extraño. Pero una vez que se iluminó el camino y tras saltar el primer obstáculo, fue un cómodo paseo. Creo que lo que trato de decir es que Haskell es genial y si estás interesado en la programación deberías aprenderlo incluso aunque te sea totalmente extraño. Aprender Haskell es como aprender a programar por primera vez ¡Es divertido! Te fuerza a que pienses diferente, lo cual nos lleva a la siguiente sección...

Entonces, ¿qué es Haskell?

Haskell es un **lenguaje de programación puramente funcional**. En los lenguajes imperativos obtenemos resultados dándole al computador una secuencia de tareas que luego éste ejecutará. Mientras las ejecuta, puede cambiar de estado. Por ejemplo, establecemos la variable **`a`** a 5, realizamos algunas tareas y luego cambiamos el valor de la variable anterior. Estos lenguajes poseen estructuras de control de flujo para realizar ciertas acciones varias veces (**`for`**, **`while`**...). Con la programación puramente funcional no decimos al computador lo que tiene que hacer, sino más bien, decimos como son las cosas. El factorial de un número es el producto de todos los números desde el 1 hasta ese número, la suma de una lista de números es el primer número más la suma del resto de la lista, etc. Expresamos la forma de las funciones. Además no podemos establecer una variable a algo y luego establecerla a otra cosa. Si decimos que **`a`** es 5, luego no podemos decir que es otra cosa porque acabamos de decir que es 5 ¿Acaso somos unos mentirosos? De este modo, en los lenguajes puramente funcionales, una función no tiene efectos secundarios. Lo único que

puede hacer una función es calcular y devolver algo como resultado. Al principio esto puede parecer una limitación pero en realidad tiene algunas buenas consecuencias: si una función es llamada dos veces con los mismos parámetros, obtendremos siempre el mismo resultado. A esto lo llamamos *transparencia referencial* y no solo permite al compilador razonar acerca de el comportamiento de un programa, sino que también nos permite deducir fácilmente (e incluso demostrar) que una función es correcta y así poder construir funciones más complejas uniendo funciones simples.

Haskell es **perezoso**. Es decir, a menos que le indiquemos lo contrario, Haskell no ejecutará funciones ni calculará resultados hasta que se vea realmente forzado a hacerlo. Esto funciona muy bien junto con la transparencia referencial y permite que veamos los programas como una serie de transformaciones de datos. Incluso nos permite hacer cosas interesantes como estructuras de datos infinitas. Digamos que tenemos una lista de números inmutables `xs = [1,2,3,4,5,6,7,8]` y una función `doubleMe` que multiplica cada elemento por 2 y devuelve una nueva lista. Si quisiéramos multiplicar nuestra lista por 8 en un lenguaje imperativo he hiciéramos `doubleMe(doubleMe(doubleMe(xs)))`, probablemente el computador recorrería la lista, haría una copia y devolvería el valor. Luego, recorrería otras dos veces más la lista y devolvería el valor final. En un lenguaje perezoso, llamar a `doubleMe` con una lista sin forzar que muestre el valor acaba con un programa diciéndote “Claro claro, ¡luego lo hago!”. Pero cuando quieres ver el resultado, el primer `doubleMe` dice al segundo que quiere el resultado, ¡ahora! El segundo dice al tercero eso mismo y éste a regañadientes devuelve un 1 duplicado, lo cual es un 2. El segundo lo recibe y devuelve un 4 al primero. El primero ve el resultado y dice que el primer elemento de la lista es un 8. De este modo, el computador solo hace un recorrido a través de la lista y solo cuando lo necesitamos. Cuando queremos calcular algo a partir de unos datos iniciales en un lenguaje perezoso, solo tenemos que tomar estos datos e ir transformándolos y moldeándolos hasta que se parezcan al resultado que deseamos.

Haskell es un lenguaje **tipificado estáticamente**. Cuando compilamos un programa, el compilador sabe que trozos del código son enteros, cuales son cadenas de texto, etc. Gracias a esto un montón de posibles errores son capturados en tiempo de compilación. Si intentamos sumar un número y una cadena de texto, el compilador nos regañará. Haskell usa un fantástico sistema de tipos que posee inferencia de tipos. Esto significa que no tenemos que etiquetar cada trozo de código explícitamente con un tipo porque el sistema de tipos lo puede deducir de forma inteligente. La inferencia de tipos también permite que nuestro código sea más general, si hemos creado una función que toma dos números y los suma y no establecemos explícitamente sus tipos, la función aceptará cualquier par de parámetros que actúen como números.

Haskell es elegante y conciso. Se debe a que utiliza conceptos de alto nivel. Los programas Haskell son normalmente más cortos que los equivalentes imperativos. Y los programas cortos son más fáciles de mantener que los largos, además de que poseen menos errores.

Haskell fue creado por unos tipos muy inteligentes (todos ellos con sus respectivos doctorados). El proyecto de crear Haskell comenzó en 1987 cuando un comité de investigadores se pusieron de acuerdo para diseñar un lenguaje revolucionario. En el 2003 el informe Haskell fue publicado, definiendo así una versión estable del lenguaje.

Qué necesitas para comenzar

Un editor de texto y un compilador de Haskell. Probablemente ya tienes instalado tu editor de texto favorito así que no vamos a perder el tiempo con esto. Ahora mismo, los dos principales compiladores de Haskell son GHC (Glasgow Haskell Compiler) y Hugs. Para los propósitos de esta guía usaremos GHC. No voy a cubrir muchos detalles de la instalación. En Windows es cuestión de descargarse el instalador, pulsar “siguiente” un par de veces y luego reiniciar el ordenador. En las distribuciones de Linux basadas en Debian se puede instalar con `apt-get` o instalando un paquete `deb`. En MacOS es cuestión de instalar un `dmg`

o utilizar **`macports`**. Sea cual sea tu plataforma, aquí tienes más información.

GHC toma un script de Haskell (normalmente tienen la extensión *.hs*) y lo compila, pero también tiene un modo interactivo el cual nos permite interactuar con dichos scripts. Podemos llamar a las funciones de los scripts que hayamos cargado y los resultados serán mostrados de forma inmediata. Para aprender es mucho más fácil y rápido en lugar de tener que compilar y ejecutar los programas una y otra vez. El modo interactivo se ejecuta tecleando **`ghci`** desde tu terminal. Si hemos definido algunas funciones en un fichero llamado, digamos, **`misFunciones.hs`**, podemos cargar esas funciones tecleando **`:l misFunciones`**, siempre y cuando **`misFunciones.hs`** esté en el mismo directorio en el que fue invocado **`ghci`**. Si modificamos el script *.hs* y queremos observar los cambios tenemos que volver a ejecutar **`:l misFunciones`** o ejecutar **`:r`** que es equivalente ya que recarga el script actual. Trabajaremos definiendo algunas funciones en un fichero *.hs*, las cargamos y pasamos el rato jugando con ellas, luego modificaremos el fichero *.hs* volviendo a cargarlo y así sucesivamente. Seguiremos este proceso durante toda la guía.

Empezando

¡Preparados, listos, ya!

Muy bien ¡Vamos a empezar! Si eres esa clase de mala persona que no lee las introducciones y te la has saltado, quizás debas leer la última sección de la introducción porque explica lo que necesitas para seguir esta guía y como vamos a trabajar. Lo primero que vamos a hacer es ejecutar GHC en modo interactivo y utilizar algunas funciones para ir acostumbrándonos un poco. Abre una terminal y escribe **`ghci`**. Serás recibido con un saludo como éste:

```
GHCi, version 7.2.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

¡Enhorabuena, entraste de GHCi! Aquí el apuntador (o *prompt*) es **`Prelude>`** pero como éste se hace más largo a medida que cargamos módulos durante una sesión, vamos a utilizar **`ghci>`**. Si quieres tener el mismo apuntador ejecuta **`:set prompt "ghci> "`**.

Aquí tenemos algo de aritmética simple.

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
ghci> 5 / 2
2.5
ghci>
```

Se explica por si solo. También podemos utilizar varias operaciones en una misma línea de forma que se sigan todas las reglas de precedencia que todos conocemos. Podemos usar paréntesis para utilizar una precedencia explícita.

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

¿Muy interesante, eh? Sí, se que no, pero ten paciencia. Una pequeña dificultad a tener en cuenta ocurre cuando negamos números, siempre será mejor rodear los números negativos con paréntesis. Hacer algo como **`5 * -3`** hará que GHCi se enfade, sin embargo **`5 * (-3)`** funcionará.

La álgebra booleana es también bastante simple. Como seguramente sabrás, **&&** representa el **Y** lógico mientras que **||** representa el **O** lógico. **not** niega **True** a **False** y viceversa.

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

La comprobación de igualdad se hace así:

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hola" == "hola"
True
```

¿Qué pasa si hacemos algo como **5 + "texto"** o **5 == True**? Bueno, si probamos con el primero obtenemos este amigable mensaje de error:

```
No instance for (Num [Char])
arising from a use of `+' at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "texto"
In the definition of `it': it = 5 + "texto"
```

GHCi nos está diciendo es que **"texto"** no es un número y por lo tanto no sabe como sumarlo a 5. Incluso si en lugar de **"texto"** fuera **"cuatro"**, **"four"**, o **"4"**, Haskell no lo consideraría como un número. **+** espera que su parte izquierda y derecha sean números. Si intentamos realizar **True == 5**, GHCi nos diría que los tipos no coinciden. Mientras que **+** funciona solo con cosas que son consideradas números, **==** funciona con cualquiera cosa que pueda ser comparada. El truco está en que ambas deben ser comparables entre si. No podemos comparar la velocidad con el tocino. Daremos un vistazo más detallado sobre los tipos más adelante. Nota: podemos hacer **5 + 4.0** porque **5** no posee un tipo concreto y puede actuar como un entero o como un número en coma flotante. **4.0** no puede actuar como un entero, así que **5** es el único que se puede adaptar.

Puede que no lo sepas, pero hemos estado usando funciones durante todo este tiempo. Por ejemplo, ***** es una función que toma dos números y los multiplica. Como ya has visto, lo llamamos haciendo un sándwich sobre él. Esto lo llamamos funciones infijas. Muchas funciones que no son usadas con números son prefijas. Vamos a ver alguna de ellas.

Las funciones normalmente son prefijas así que de ahora en adelante no vamos a decir que una función está en forma prefija, simplemente lo asumiremos. En muchos lenguajes imperativos las funciones son llamadas escribiendo su nombre y luego escribiendo sus parámetros entre paréntesis, normalmente separados por comas. En Haskell, las funciones son llamadas escribiendo su nombre, un espacio y sus parámetros, separados por espacios. Para empezar, vamos a intentar llamar a una de las funciones más aburridas de Haskell.

```
ghci> succ 8
9
```

La función **succ** toma cualquier cosa que tenga definido un sucesor y devuelve ese sucesor. Como puedes ver, simplemente hemos separado el nombre de la función y su parámetro por un espacio. Llamar a una función con varios parámetros es igual de sencillo. Las funciones **min** y **max** toman dos cosas que puedan ponerse en orden (¡cómo los números!) y devuelven uno de ellos.

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

La aplicación de funciones (llamar a una función poniendo un espacio después de ella y luego escribir sus parámetros) tiene la máxima prioridad. Dicho con un ejemplo, estas dos sentencias son equivalentes:

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

Sin embargo, si hubiésemos querido obtener el sucesor del producto de los números 9 y 10, no podríamos haber escrito **succ 9 * 10** porque hubiésemos obtenido el sucesor de 9, el cual hubiese sido multiplicado por 10, obteniendo 100. Tenemos que escribir **succ (9 * 10)** para obtener 91.

Si una función toma dos parámetros también podemos llamarla como una función infija rodeándola con acentos abiertos. Por ejemplo, la función **div** toma dos enteros y realiza una división entera entre ellos. Haciendo **div 92 10** obtendríamos 9. Pero cuando la llamamos así, puede haber alguna confusión como que número está haciendo la división y cual está siendo dividido. De manera que nosotros la llamamos como una función infija haciendo **92 `div` 10**, quedando de esta forma más claro.

La gente que ya conoce algún lenguaje imperativo tiende a aferrarse a la idea de que los paréntesis indican una aplicación de funciones. Por ejemplo, en C, usas los paréntesis para llamar a las funciones como **foo()**, **bar(1)**, o **baz(3, "jaja")**. Como hemos dicho, los espacios son usados para la aplicación de funciones en Haskell. Así que estas funciones en Haskell serían **foo**, **bar 1** y **baz 3 "jaja"**. Si ves algo como **bar (bar 3)** no significa que **bar** es llamado con **bar** y **3** como parámetros. Significa que primero llamamos a la función **bar** con **3** como parámetro para obtener un número y luego volver a llamar **bar** otra vez con ese número. En C, esto sería algo como **bar(bar(3))**.

Las primeras pequeñas funciones

En la sección anterior obtuvimos una idea básica de como llamar a las funciones ¡Ahora vamos a intentar hacer las nuestras! Abre tu editor de textos favorito y pega esta función que toma un número y lo multiplica por dos.

```
doubleMe x = x + x
```

Las funciones son definidas de forma similar a como son llamadas. El nombre de la función es seguido por los parámetros separados por espacios. Pero, cuando estamos definiendo funciones, hay un **=** y luego definimos lo que hace la función. Guarda esto como **baby.hs** o como tú quieras. Ahora navega hasta donde lo guardaste y ejecuta **ghci** desde ahí. Una vez dentro de GHCi, escribe **:l baby**. Ahora que nuestro código está cargado, podemos jugar con la función que hemos definido.

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

Como **+** funciona con los enteros igual de bien que con los número en coma flotante (en realidad con cualquier cosa que pueda ser considerada un número), nuestra función también funciona con cualquier número. Vamos a hacer una función que tome dos números, multiplique por dos cada uno de ellos y luego sume ambos.

```
doubleUs x y = x*2 + y*2
```

Simple. La podríamos haber definido también como **`doubleUs x y = x + x + y + y`**. Ambas formas producen resultados muy predecibles (recuerda añadir esta función en el fichero **`baby.hs`**, guardarlo y luego ejecutar **`:l baby`** dentro de GHCi).

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

Como podrás deducir, puedes llamar tus propias funciones dentro de las funciones que hagas. Teniendo esto en cuenta, podríamos redefinir **`doubleUs`** como:

```
doubleUs x y = doubleMe x + doubleMe y
```

Esto es un simple ejemplo de un patrón normal que verás por todo Haskell. Crear funciones pequeñas que son obviamente correctas y luego combinarlas en funciones más complejas. De esta forma también evitarás repetirte. ¿Qué pasa si algunos matemáticos descubren que 2 es en realidad 3 y tienes que cambiar tu programa? Puedes simplemente redefinir **`doubleMe`** para que sea **`x + x + x`** y como **`doubleUs`** llama a **`doubleMe`** automáticamente funcionara en este extraño mundo en el que 2 es 3.

Las funciones en Haskell no tienen que estar en ningún orden en particular, así que no importa si defines antes **`doubleMe`** y luego **`doubleUs`** o si lo haces al revés.

Ahora vamos a crear una función que multiplique un número por 2 pero solo si ese número es menor o igual que 100, porque los número mayores 100 ya son suficientemente grandes por si solos.

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```

Acabamos de introducir la sentencia **`if`** de Haskell. Probablemente ya estés familiarizado con la sentencia **`if`** de otros lenguajes. La diferencia entre la sentencia **`if`** de Haskell y la de los lenguajes imperativos es que la parte **`else`** es obligatoria. En los lenguajes imperativos podemos saltarnos unos cuantos pasos si una condición no se ha satisfecho pero en Haskell cada expresión o función debe devolver un valor. También podríamos haber definido la sentencia **`if`** en una sola línea pero así parece un poco mas legible. Otro asunto acerca de la sentencia **`if`** en Haskell es que es una expresión. Básicamente una expresión es un trozo de código que devuelve un valor. **`5`** es una expresión porque devuelve 5, **`4 + 8`** es una expresión, **`x + y`** es una expresión porque devuelve la suma de **`x`** e **`y`**.

Como la parte **else** es obligatoria, una sentencia **if** siempre devolverá algo y por tanto es una expresión. Si queremos sumar uno a cada número que es producido por la función anterior, podemos escribir su cuerpo así.

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

Si hubiésemos omitido los paréntesis, sólo hubiera sumado uno si **x** no fuera mayor que 100. Fíjate en el **'** al final del nombre de la función. Ese apóstrofe no tiene ningún significado especial en la sintaxis de Haskell. Es un carácter válido para ser usado en el nombre de una función. Normalmente usamos **'** para denotar la versión estricta de una función (una que no es perezosa) o una pequeña versión modificada de una función o variable. Como **'** es un carácter válido para la funciones, podemos hacer cosas como esta.

```
conanO'Brien = "¡Soy yo, Conan O'Brien!"
```

Hay dos cosas que nos quedan por destacar. La primera es que el nombre de esta función no empieza con mayúsculas. Esto se debe a que las funciones no pueden empezar con una letra en mayúsculas. Veremos el porqué un poco más tarde. La segunda es que esta función no toma ningún parámetro, normalmente lo llamamos una definición (o un nombre). Como no podemos cambiar las definiciones (y las funciones) después de que las hayamos definido, **conanO'Brien** y la cadena **"It's a-me, Conan O'Brien!"** se pueden utilizar indistintamente.

Una introducción a las listas

Al igual que las listas de la compra de la vida real, las listas en Haskell son muy útiles. Es la estructura de datos más utilizada y pueden ser utilizadas de diferentes formas para modelar y resolver un montón de problemas. Las listas son MUY importantes. En esta sección daremos un vistazo a las bases sobre las listas, cadenas de texto (las cuales son listas) y listas intensionales.

En Haskell, las listas son una estructura de datos **homogénea**. Almacena varios elementos del mismo tipo. Esto significa que podemos crear una lista de enteros o una lista de caracteres, pero no podemos crear una lista que tenga unos cuantos enteros y otros cuantos caracteres. Y ahora, ¡una lista!

Nota

Podemos usar la palabra reservada **let** para definir un nombre en GHCi. Hacer **let a = 1** dentro de GHCi es equivalente ha escribir **a = 1** en un fichero y luego cargarlo.

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

Como puedes ver, las listas se definen mediante corchetes y sus valores se separan por comas. Si intentáramos crear una lista como esta **[1,2,'a',3,'b','c',4]**, Haskell nos avisaría que los caracteres (que por cierto son declarados como un carácter entre comillas simples) no son números. Hablando sobre caracteres, las cadenas son simplemente listas de caracteres. **"hello"** es solo una alternativa sintáctica de **['h','e','l','l','o']**. Como las cadenas son listas, podemos usar las funciones que operan con listas sobre ellas, lo cual es realmente útil.

Una tarea común es concatenar dos listas. Cosa que conseguimos con el operador **++**.

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
```

```
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

Hay que tener cuidado cuando utilizamos el operador **++** repetidas veces sobre cadenas largas. Cuando concatenamos dos listas (incluso si añadimos una lista de un elemento a otra lista, por ejemplo **[1,2,3] ++ [4]**, internamente, Haskell tiene que recorrer la lista entera desde la parte izquierda del operador **++**. Esto no supone ningún problema cuando trabajamos con listas que no son demasiado grandes. Pero concatenar algo al final de una lista que tiene cincuenta millones de elementos llevará un rato. Sin embargo, concatenar algo al principio de una lista utilizando el operador **:** (también llamado operador cons) es instantáneo.

```
ghci> 'U':"n gato negro"
"Un gato negro"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

Fíjate que **:** toma un número y una lista de números o un carácter y una lista de caracteres, mientras que **++** toma dos listas. Incluso si añades un elemento al final de las lista con **++**, hay que rodearlo con corchetes para que se convierte en una lista de un solo elemento.

```
ghci> [1,2] ++ 3
<interactive>:1:10:
    No instance for (Num [a0])
      arising from the literal `3'
      [...]

ghci> [1,2] ++ [3]
[1,2,3]
```

[1,2,3] es una alternativa sintáctica de **1:2:3:[]**. **[]** es una lista vacía. Si anteponemos 3 a ella con **:**, obtenemos **[3]**, y si anteponemos 2 a esto obtenemos **[2,3]**.

Nota

[], **[[]]** y **[[],[],[]]** son cosas diferentes entre si. La primera es una lista vacía, la segunda es una lista que contiene un elemento (una lista vacía) y la tercera es una lista que contiene tres elementos (tres listas vacías).

Si queremos obtener un elemento de la lista sabiendo su índice, utilizamos **!!**. Los índices empiezan por 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

Pero si intentamos obtener el sexto elemento de una lista que solo tiene cuatro elementos, obtendremos un error, así que hay que ir con cuidado.

Las listas también pueden contener listas. Estas también pueden contener a su vez listas que contengan listas, que contengan listas...

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
```

```
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

Las listas dentro de las listas pueden tener diferentes tamaños pero no pueden tener diferentes tipos. De la misma forma que no se puede contener caracteres y números en un lista, tampoco se puede contener listas que contengan listas de caracteres y listas de números.

Las listas pueden ser comparadas si los elementos que contienen pueden ser comparados. Cuando usamos **<**, **<=**, **>**, y **>=** para comparar listas, son comparadas en orden lexicográfico. Primero son comparadas las cabezas. Luego son comparados los segundos elementos y así sucesivamente.

¿Qué mas podemos hacer con las listas? Aquí tienes algunas funciones básicas que pueden operar con las listas.

- `head` toma una lista y devuelve su cabeza. La cabeza de una lista es básicamente el primer elemento.

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

- `tail` toma una lista y devuelve su cola. En otros palabras, corta la cabeza de la lista.

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

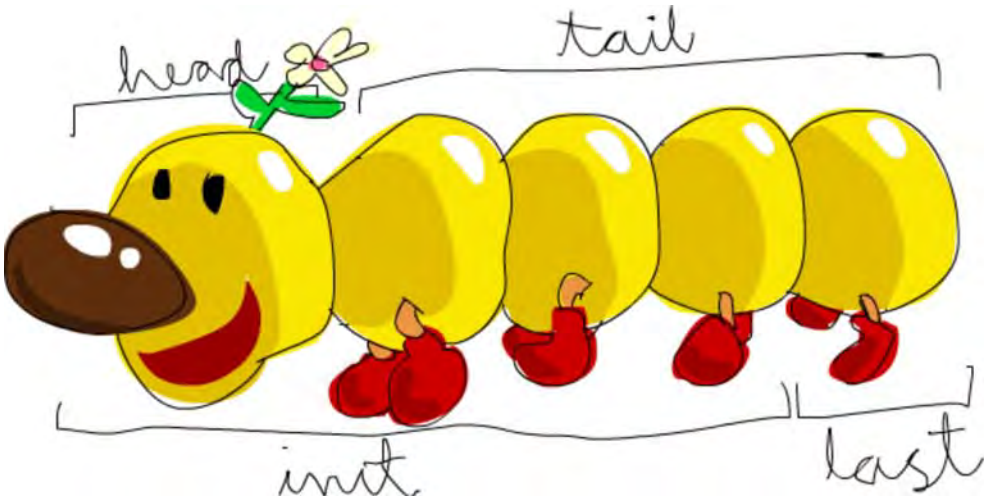
- `last` toma una lista y devuelve su último elemento.

```
ghci> last [5,4,3,2,1]
1
```

- `init` toma una lista y devuelve toda la lista excepto su último elemento.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

Si imaginamos las listas como monstruos, serian algo como:



¿Pero que pasa si intentamos obtener la cabeza de una lista vacía?

```
ghci> head []
*** Exception: Prelude.head: empty list
```

¡Oh, lo hemos roto! Si no hay monstruo, no hay cabeza. Cuando usamos **`head`**, **`tail`**, **`last`** e **`init`** debemos tener precaución de no usar con ellas listas vacías. Este error no puede ser capturado en tiempo de compilación así que siempre es una buena práctica tomar precauciones antes de decir a Haskell que te devuelva algunos elementos de una lista vacía.

- `length` toma una lista y obviamente devuelve su tamaño.

  ```
  ghci> length [5,4,3,2,1]
  5
  ```

- `null` comprueba si una lista está vacía. Si lo está, devuelve **`True`**, en caso contrario devuelve **`False`**. Usa esta función en lugar de **`xs == []`** (si tienes una lista que se llame xs).

  ```
  ghci> null [1,2,3]
  False
  ghci> null []
  True
  ```

- `reverse` pone del revés una lista.

  ```
  ghci> reverse [5,4,3,2,1]
  [1,2,3,4,5]
  ```

- `take` toma un número y una lista y extrae dicho número de elementos de una lista. Observa.

  ```
  ghci> take 3 [5,4,3,2,1]
  [5,4,3]
  ghci> take 1 [3,9,3]
  [3]
  ghci> take 5 [1,2]
  [1,2]
  ghci> take 0 [6,6,6]
  []
  ```

  Fíjate que si intentamos tomar más elementos de los que hay en una lista, simplemente devuelve la lista. Si tomamos 0 elementos, obtenemos una lista vacía.

- `drop` funciona de forma similar, solo que quita un número de elementos del comienzo de la lista.

  ```
  ghci> drop 3 [8,4,2,1,5,6]
  [1,5,6]
  ghci> drop 0 [1,2,3,4]
  [1,2,3,4]
  ghci> drop 100 [1,2,3,4]
  []
  ```

- `maximum` toma una lista de cosas que se pueden poner en algún tipo de orden y devuelve el elemento más grande.

- `minimum` devuelve el más pequeño.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

- `sum` toma una lista de números y devuelve su suma.

- `product` toma una lista de números y devuelve su producto.

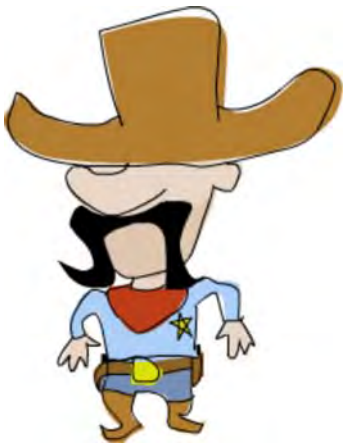
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

- `elem` toma una cosa y una lista de cosas y nos dice si dicha cosa es un elemento de la lista. Normalmente, esta función es llamada de forma infija porque resulta más fácil de leer.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

Estas fueron unas cuantas funciones básicas que operan con listas. Veremos más funciones que operan con listas más adelante.

Texas rangos



¿Qué pasa si queremos una lista con todos los números entre el 1 y el 20? Sí, podríamos simplemente escribirlos todos pero obviamente esto no es una solución para los que buscan buenos lenguajes de programación. En su lugar, usaremos rangos. Los rangos son una manera de crear listas que contengan una secuencia aritmética de elementos enumerables. Los números pueden ser enumerados. Uno, dos, tres, cuatro, etc. Los caracteres también pueden ser enumerados. El alfabeto es una enumeración de caracteres desde la A hasta la Z. Los nombres no son enumerables. ¿Qué viene después de “Juan”? Ni idea.

Para crear una lista que contenga todos los números naturales desde el 1 hasta el 20 simplemente escribimos **`[1..20]`**. Es equivalente a escribir **`[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]`** y no hay ninguna diferencia entre escribir uno u otro salvo que escribir manualmente una larga secuencia de enumerables es bastante estúpido.

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

También podemos especificar el número de pasos entre elementos de un rango ¿Y si queremos todos los números pares desde el 1 hasta el 20? ¿o cada tercer número?

```
ghci> [2,4..20]
```

```
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Es cuestión de separar los primeros dos elementos con una coma y luego especificar el límite superior. Aunque son inteligentes, los rangos con pasos no son tan inteligentes como algunas personas esperan que sean. No puedes escribir **`[1,2,4,8,16..100]`** y esperar obtener todas las potencias de 2. Primero porque solo se puede especificar un paso. Y segundo porque las secuencias que no son aritméticas son ambiguas si solo damos unos pocos elementos iniciales.

Para obtener una lista con todos los números desde el 20 hasta el 1 no podemos usar **`[20..1]`**, debemos utilizar **`[20,19..1]`**.

¡Cuidado cuando uses números en coma flotante con los rangos! Éstos no son del todo precisos (por definición), y su uso con los rangos puede dar algunos resultados no esperados.

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Mi consejo es no utilizar rangos con números en coma flotante.

También podemos utilizar los rangos para crear listas infinitas simplemente no indicando un límite superior. Más tarde nos centraremos más en las listas infinitas. Por ahora, vamos a examinar como obtendríamos los primeros 24 múltiplos de 13. Sí, podemos utilizar **`[13,26..24*13]`**. Pero hay una forma mejor: **`take 13 [13,26..]`**. Como Haskell es perezoso, no intentará evaluar la lista infinita inmediatamente porque no terminaría nunca. Esperará a ver que es lo que quieres obtener de la lista infinita. En este caso ve que solo queremos los primeros 24 elementos y los evalúa con mucho gusto.

Ahora, un par de funciones que generan listas infinitas:

- `cycle` toma una lista y crea un ciclo de listas iguales infinito. Si intentáramos mostrar el resultado nunca terminaría así que hay que cortarlo en alguna parte.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

- `repeat` toma un elemento y produce una lista infinita que contiene ese único elemento repetido. Es como hacer un ciclo de una lista con un solo elemento.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

Aunque aquí sería más simple usar la función `replicate`, ya que sabemos el número de elementos de antemano. **`replicate 3 10`** devuelve **`[10,10,10]`**.

Soy una lista intensional

Si alguna vez tuviste clases de matemáticas, probablemente viste algún conjunto definido de forma intensiva, definido a partir de otros conjuntos más generales. Un conjunto definido de forma intensiva que contenga los diez primeros números naturales pares sería $S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$. La parte anterior al separador se llama la función de salida,

x es la variable, $\mathbb{N}$ es el conjunto de entrada y $x \leq 10$ es el predicado. Esto significa que el conjunto contiene todos los dobles de los número naturales que cumplen el predicado.

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

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

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

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

Bien, funciona. ¿Y si quisiéramos todos los números del 50 al 100 cuyo resto al dividir por 7 fuera 3? Fácil:

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

¡Todo un éxito! Al hecho de eliminar elementos de la lista utilizando predicados también se conoce como **filtrado**. Tomamos una lista de números y la filtramos usando predicados. Otro ejemplo, digamos que queremos lista intensional que reemplace cada número impar mayor que diez por “BANG!” y cada número impar menor que diez por “BOOM!”. Si un número no es impar, lo dejamos fuera de la lista. Para mayor comodidad, vamos a poner la lista intensional dentro de una función para que sea fácilmente reutilizable.

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

La última parte de la comprensión es el predicado. La función **`odd`** devuelve **`True`** si le pasamos un número impar y **`False`** con uno par. El elemento es incluido en la lista solo si todos los predicados se evalúan a **`True`**.

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

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

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

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

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

Como era de esperar, la longitud de la nueva lista es de 9 ¿Y si quisiéramos todos los posibles productos cuyo valor sea mayor que 50?

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

¿Qué tal una lista intensional que combine una lista de adjetivos con una lista de nombres? Solo para quedarnos tranquilos...

```
ghci> let nouns = ["rana","zebra","cabra"]
ghci> let adjetives = ["perezosa","enfadada","intrigante"]
ghci> [noun ++ " " ++ adjetive | noun <- nouns, adjetive <- adjetives]
["rana perezosa","rana enfadada","rana intrigante","zebra perezosa",
"zebra enfadada","zebra intrigante","cabra perezosa","cabra enfadada",
"cabra intrigante"]
```

¡Ya se! Vamos a escribir nuestra propia versión de **`length`**. La llamaremos **`length'`**.

```
length' xs = sum [1 | _ <- xs]
```

_ significa que no nos importa lo que vayamos a extraer de la lista, así que en vez de escribir el nombre de una variable que nunca usaríamos, simplemente escribimos **_**. La función reemplaza cada elemento de la lista original por 1 y luego los suma. Esto significa que la suma resultante será el tamaño de nuestra lista.

Un recordatorio: como las cadenas son listas, podemos usar las listas intensionales para procesar y producir cadenas. Por ejemplo, una función que toma cadenas y elimina de ellas todo excepto las letras mayúsculas sería algo tal que así:

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Unas pruebas rápidas:

```
ghci> removeNonUppercase "Jajaja! Ajajaja!"
"JA"
ghci> removeNonUppercase "noMEGUSTANLASRANAS"
"MEGUSTANLASRANAS"
```

En este caso el predicado hace todo el trabajo. Dice que el elemento será incluido en la lista solo si es un elemento de **`[A..Z]`**. Es posible crear listas intensionales anidadas si estamos trabajando con listas que contienen listas. Por ejemplo, dada una lista de listas de números, vamos eliminar los números impares sin aplanar la lista:

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Podemos escribir las listas intensionales en varias líneas. Si no estamos usando GHCi es mejor dividir las listas intensionales en varias líneas, especialmente si están anidadas.

Tuplas

De alguna forma, las tuplas son parecidas a las listas. Ambas son una forma de almacenar

varios valores en un solo valor. Sin embargo, hay unas cuantas diferencias fundamentales. Una lista de números es una lista de números. Ese es su tipo y no importa si tiene un sólo elemento o una cantidad infinita de ellos. Las tuplas sin embargo, son utilizadas cuando sabes exactamente cuantos valores tienen que ser combinados y su tipo depende de cuantos componentes tengan y del tipo de estos componentes. Las tuplas se denotan con paréntesis y sus valores se separan con comas.

Otra diferencia clave es que no tienen que ser homogéneas. Al contrario que las listas, las tuplas pueden contener una combinación de valores de distintos tipos.

Piensa en como representaríamos un vector bidimensional en Haskell. Una forma sería utilizando listas. Podría funcionar. Entonces, ¿si quisiéramos poner varios vectores dentro de una lista que representa los puntos de una figura bidimensional? Podríamos usar algo como **`[[1,2],[8,11],[4,5]]`**. El problema con este método es que también podríamos hacer cosas como **`[[1,2],[8,11,5],[4,5]]`** ya que Haskell no tiene problemas con ello, sigue siendo una lista de listas de números pero no tiene ningún sentido. Pero una tupla de tamaño 2 (también llamada dupla) tiene su propio tipo, lo que significa que no puedes tener varias duplas y una tripla (una tupla de tamaño 3) en una lista, así que vamos a usar éstas. En lugar de usar corchetes rodeando los vectores utilizamos paréntesis: **`[(1,2),(8,11),(4,5)]`**. ¿Qué pasaría si intentamos crear una forma como **`[(1,2),(8,11,5),(4,5)]`**? Bueno, obtendríamos este error:

```
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Nos está diciendo que hemos intentado usar una dupla y una tripla en la misma lista, lo cual no esta permitido ya que las listas son homogéneas y un dupla tiene un tipo diferente al de una tripla (aunque contengan el mismo tipo de valores). Tampoco podemos hacer algo como **`[(1,2),("uno",2)]`** ya que el primer elemento de la lista es una tupla de números y el segundo es una tupla de una cadena y un número. Las tuplas pueden ser usadas para representar una gran variedad de datos. Por ejemplo, si queremos representar el nombre y la edad de alguien en Haskell, podemos utilizar la tripla: **`("Christopher", "Walken", 55)`**. Como hemos visto en este ejemplo las tuplas también pueden contener listas.

Utilizamos la tuplas cuando sabemos de antemano cuantos componentes de algún dato debemos tener. Las tuplas son mucho más rígidas que las listas ya que para cada tamaño tienen su propio tipo, así que no podemos escribir una función general que añada un elemento a una tupla: tenemos que escribir una función para añadir duplas, otra función para añadir triplas, otra función para añadir cuádruplas, etc.

Mientras que existen listas unitarias, no existen tuplas unitarias. Realmente no tiene mucho sentido si lo piensas. Una tupla unitaria sería simplemente el valor que contiene y no nos aportaría nada útil.

Como las listas, las tuplas pueden ser comparadas si sus elementos pueden ser comparados. Solo que no podemos comparar dos tuplas de diferentes tamaños mientras que si podemos comparar dos listas de diferentes tamaños. Dos funciones útiles para operar con duplas son:

- `fst` toma una dupla y devuelve su primer componente.

```
ghci> fst (8,11)
8
ghci> fst ("Wow", False)
"Wow"
```

- `snd` toma una dupla y devuelve su segundo componente. ¡Sorpresa!

```
ghci> snd (8,11)
11
ghci> snd ("Wow", False)
False
```

Nota

Estas funciones solo operan sobre duplas. No funcionaran sobre triplas, cuádruplas, quíntuplas, etc. Veremos más formas de extraer datos de las tuplas un poco más tarde.

Ahora una función interesante que produce listas de duplas es `zip`. Esta función toma dos listas y las une en un lista uniendo sus elementos en una dupla. Es una función realmente simple pero tiene montones de usos. Es especialmente útil cuando queremos combinar dos listas de alguna forma o recorrer dos listas simultáneamente. Aquí tienes una demostración:

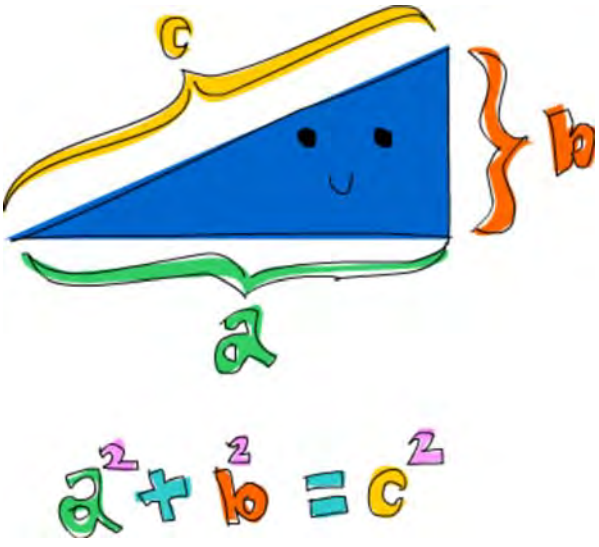
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["uno","dos","tres","cuatro","cinco"]
[(1,"uno"),(2,"dos"),(3,"tres"),(4,"cuatro"),(5,"cinco")]
```

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

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

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

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



He aquí un problema que combina tuplas con listas intensionales: ¿Qué triángulo recto cuyos lados miden enteros menores que 10 tienen un perímetro igual a 24? Primero, vamos a intentar generar todos los triángulos con lados iguales o menores que 10:

```
ghci> let triangles = [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]
```

Simplemente estamos extrayendo valores de estas tres listas y nuestra función de salida las esta combinando en una tripla. Si evaluamos esto escribiendo **`triangles`** en GHCi, obtendremos una lista con todos los posibles triángulos cuyos lados son menores o iguales que 10. Ahora, debemos añadir una condición que nos filtre únicamente los triángulos rectos. Vamos a modificar esta función teniendo en consideración que el lado b no es mas largo que la hipotenusa y que el lado a no es más largo que el lado b.

```
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2
```

Ya casi hemos acabado. Ahora, simplemente modificaremos la función diciendo que solo queremos aquellos que su perímetro es 24.

```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2
ghci> rightTriangles'
[(6,8,10)]
```

¡Y ahí está nuestra respuesta! Este método de resolución de problemas es muy común en la programación funcional. Empiezas tomando un conjunto de soluciones y vas aplicando transformaciones para ir obteniendo soluciones, filtrándolas una y otra vez hasta obtener las soluciones correctas.

Tipos y clases de tipos

Cree en el tipo

Anteriormente mencionamos que Haskell tiene un sistema de tipos estático. Se conoce el tipo de cada expresión en tiempo de compilación, lo que produce código más seguro. Si escribimos un programa que intenta dividir un valor del tipo booleano por un número, no llegará a compilarse. Esto es bueno ya que es mejor capturar este tipo de errores en tiempo de compilación en lugar de que el programa falle. Todo en Haskell tiene un tipo, de forma que el compilador puede razonar sobre el programa antes de compilarlo.

Al contrario que Java o C, Haskell posee inferencia de tipos. Si escribimos un número, no tenemos que especificar que eso es un número. Haskell puede deducirlo él solo, así que no tenemos que escribir explícitamente los tipos de nuestras funciones o expresiones para conseguir resultados. Ya hemos cubierto parte de las bases de Haskell con muy poco conocimiento de los tipos. Sin embargo, entender el sistema de tipos es una parte muy importante para dominar Haskell.

Un tipo es como una etiqueta que posee toda expresión. Esta etiqueta nos dice a que categoría de cosas se ajusta la expresión. La expresión **`True`** es un booleano, **`"Hello"`** es una cadena, etc.

Ahora vamos a usar GHCi para examinar los tipos de algunas expresiones. Lo haremos gracias al comando **`:t`**, el cual, seguido de una expresión válida nos dice su tipo. Vamos a dar un vistazo:

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HOLA!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Podemos ver que ejecutando el comando **`:t`** sobre una expresión se muestra esa misma expresión seguida de **`::`** y de su tipo. **`::`** se puede leer como *tiene el tipo*. Los tipos explícitos siempre se escriben con su primera letra en mayúsculas. **`'a'`**, como hemos visto, tiene el tipo **`Char`**. El nombre de este tipo viene de “Character” (carácter en inglés). **`True`** tiene el tipo **`Bool`**. Tiene sentido. Pero, ¿qué es esto? Examinando el tipo de **`"HOLA!"`** obtenemos **`[Char]`**. Los corchetes definen una lista. Así que leemos esto como una *lista de caracteres*. Al contrario que las listas, cada tamaño de tupla tiene su propio tipo. Así que la expresión **`(True, 'a')`** tiene el tipo **`(Bool, Char)`**, mientras que la expresión **`('a', 'b', 'c')`** tiene el tipo **`(Char, Char, Char)`**. **`4 == 5`** siempre devolverá **`False`** así que esta expresión tiene el tipo **`Bool`**.

Las funciones también tiene tipos. Cuando escribimos nuestras propias funciones podemos darles un tipo explícito en su declaración. Generalmente está bien considerado escribir los tipos explícitamente en la declaración de un función, excepto

cuando éstas son muy cortas. De aquí en adelante les daremos tipos explícitos a todas las funciones que creemos. ¿Recuerdas la lista intensional que filtraba solo las mayúsculas de una cadena? Aquí tienes como se vería con su declaración de tipo:

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

removeNonUppercase tiene el tipo **[Char] -> [Char]**, que significa que es una función que toma una cadena y devuelve otra cadena. El tipo **[Char]** es sinónimo de **String** así que sería más elegante escribir el tipo como **removeNonUppercase :: String -> String**. Anteriormente no le dimos un tipo a esta función ya que el compilador puede inferirlo por si solo. Pero, ¿cómo escribimos el tipo de una función que toma varios parámetros? Aquí tienes una función que toma tres enteros y los suma:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Los parámetros están separados por **->** y no existe ninguna diferencia especial entre los parámetros y el tipo que devuelve la función. El tipo que devuelve la función es el último elemento de la declaración y los parámetros son los restantes. Más tarde veremos porque simplemente están separados por **->** en lugar de tener algún tipo de distinción más explícita entre los tipos de parámetros y el tipo de retorno, algo como **Int, Int, Int -> Int**.

Si escribimos una función y no tenemos claro el tipo que debería tener, siempre podemos escribir la función sin su tipo y ejecutar el comando **:t** sobre ella. Las funciones también son expresiones así que no hay ningún problema en usar **:t** con ellas.

Aquí tienes una descripción de los tipos más comunes:

- `Int` representa enteros. Se utiliza para representar número enteros, por lo que **7** puede ser un **Int** pero **7.2** no puede. **Int** está acotado, lo que significa que tiene un valor máximo y un valor mínimo. Normalmente en máquinas de 32bits el valor máximo de **Int** es 2147483647 y el mínimo -2147483648.

- `Integer` representa... esto... enteros también. La diferencia es que no están acotados así que pueden representar números muy grandes. Sin embargo, **Int** es más eficiente.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

- `Float` es un número real en coma flotante de simple precisión.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
ghci> circumference 4.0
25.132742
```

- `Double` es un número real en coma flotante de... ¡Doble precisión!.

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r


ghci> circumference' 4.0
25.132741228718345
```

- `Bool` es el tipo booleano. Solo puede tener dos valores: **`True`** o **`False`**.

- `Char` representa un carácter. Se define rodeado por comillas simples. Una lista de caracteres es una cadena.

Las tuplas también poseen tipos pero dependen de su longitud y del tipo de sus componentes, así que teóricamente existe una infinidad de tipos de tuplas y eso son demasiados tipos como para cubrirlos en esta guía. La tupla vacía es también un tipo `()` el cual solo puede contener un valor: **`()`**.

Variables de tipo

¿Cual crees que es el tipo de la función **head**? Como **head** toma una lista de cualquier tipo y devuelve su primer elemento... ¿Cual podrá ser? Vamos a verlo:

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

Hmmm... ¿Qué es **a**? ¿Es un tipo? Si recuerdas antes dijimos que los tipos deben comenzar con mayúsculas, así que no puede ser exactamente un tipo. Como no comienza con una mayúscula en realidad es una **variable de tipo**. Esto significa que **a** puede ser cualquier tipo. Es parecido a los tipos genéricos de otros lenguajes, solo que en Haskell son mucho más potentes ya que nos permite definir fácilmente funciones muy generales siempre que no hagamos ningún uso especifico del tipo en cuestión. Las funciones que tienen variables de tipos son llamadas **funciones polimórficas**. La declaración de tipo **head** representa una función que toma una lista de cualquier tipo y devuelve un elemento de ese mismo tipo.

Aunque las variables de tipo pueden tener nombres más largos de un solo carácter, normalmente les damos nombres como a, b, c, d, etc.

¿Recuerdas **`fst`**? Devuelve el primer componente de una dupla. Vamos a examinar su tipo.

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

Como vemos, **`fst`** toma una dupla que contiene dos tipos y devuelve un elemento del mismo tipo que el primer componente de la dupla. Ese es el porqué de que podamos usar **`fst`** con duplas que contengan cualquier combinación de tipos. Ten en cuenta que solo porque **a** y **b** son diferentes variables de tipo no tienen porque ser diferentes tipos. Simplemente representa que el primer componente y el valor que devuelve la función son del mismo tipo.

Clases de tipos paso a paso (1ª parte)

Las clases de tipos son una especie de interfaz que define algún tipo de comportamiento. Si un tipo es miembro de una clase de tipos, significa que ese tipo soporta e implementa el comportamiento que define la clase de tipos. La gente que viene de lenguajes orientados a objetos es propensa a confundir las clases de tipos porque piensan que son como las clases en los lenguajes orientados a objetos. Bien, pues no lo son. Una

aproximación más adecuada sería pensar que son como las interfaces de Java, o los protocolos de Objective-C, pero mejor.

¿Cuál es la declaración de tipo de la función **==**?

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

Nota

El operador de igualdad **==** es una función. También lo son **+**, **-**, *****, **/** y casi todos los operadores. Si el nombre de una función está compuesta solo por caracteres especiales (no alfanuméricos), es considerada una función infija por defecto. Si queremos examinar su tipo, pasarla a otra función o llamarla en forma prefija debemos rodearla con paréntesis. Por ejemplo: **(+) 1 4** equivale a **1 + 4**.

Interesante. Aquí vemos algo nuevo, el símbolo **=>**. Cualquier cosa antes del símbolo **=>** es una restricción de clase. Podemos leer la declaración de tipo anterior como: la función de igualdad toma dos parámetros que son del mismo tipo y devuelve un **Bool**. El tipo de estos dos parámetros debe ser miembro de la clase **Eq** (esto es la restricción de clase).

La clase de tipos **Eq** proporciona una interfaz para las comparaciones de igualdad. Cualquier tipo que tenga sentido comparar dos valores de ese tipo por igualdad debe ser miembro de la clase **Eq**. Todos los tipos estándar de Haskell excepto el tipo IO (un tipo para manejar la entrada/salida) y las funciones forman parte de la clase **Eq**.

La función **elem** tiene el tipo **(Eq a) => a -> [a] -> Bool** porque usa **==** sobre los elementos de la lista para saber si existe el elemento indicado dentro de la lista.

Algunas clases de tipos básicas son:

- `Eq` es utilizada por los tipos que soportan comparaciones por igualdad. Los miembros de esta clase implementan las funciones **==** o **/=** en algún lugar de su definición. Todos los tipos que mencionamos anteriormente forman parte de la clase **Eq** exceptuando las funciones, así que podemos realizar comparaciones de igualdad sobre ellos.

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

- `Ord` es para tipos que poseen algún orden.

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

Todos los tipos que hemos llegado a ver excepto las funciones son parte de la clase **Ord**. **Ord** cubre todas las funciones de comparación como **>**, **<**, **>=** y **<=**. La función `compare` toma dos miembros de la clase **Ord** del mismo tipo y devuelve su orden. El orden está representado por el tipo `Ordering` que puede tener tres valores distintos: **GT**, **EQ** y **LT** los cuales representan *mayor que*, *igual que* y *menor que*, respectivamente.

Para ser miembro de **Ord**, primero un tipo debe ser socio del prestigioso y exclusivo club **Eq**.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

- Los miembros de `Show` pueden ser representados por cadenas. Todos los tipos que hemos visto excepto las funciones forman parte de **Show**. la función más utilizada que trabaja con esta clase de tipos es la función `show`. Toma un valor de un tipo que pertenezca a la clase **Show** y lo representa como una cadena de texto.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

- `Read` es como la clase de tipos opuesta a **Show**. La función `read` toma una cadena y devuelve un valor del tipo que es miembro de **Read**.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Hasta aquí todo bien. Una vez más, todo los tipos que hemos visto excepto las funciones forman parte de esta clase de tipos. Pero, ¿Qué pasa si simplemente usamos **`read "4"`**?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

Lo que GHCi no está intentado decir es que no sabe que queremos que devuelva. Ten en cuenta que cuando usamos anteriormente **`read`** lo hicimos haciendo algo luego con el resultado. De esta forma, GHCi podía inferir el tipo del resultado de la función **`read`**. Si usamos el resultado de aplicar la función como un booleano, Haskell sabe que tiene que devolver un booleano. Pero ahora, lo único que sabe es que queremos un tipo de la clase **Read**, pero no cual. Vamos a echar un vistazo a la declaración de tipo de la función **`read`**.

```
ghci> :t read
read :: (Read a) => String -> a
```

¿Ves? Devuelve un tipo que es miembro de la clase **Read**, pero si luego no lo usamos en ningún otro lugar, no hay forma de saber que tipo es. Por este motivo utilizamos las **anotaciones de tipo** explícitas. Las anotación de tipo son una forma de decir explícitamente el tipo que debe tener una expresión. Lo hacemos añadiendo **`::`** al final de la expresión y luego especificando el tipo. Observa:

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

La mayoría de expresiones son del tipo que el compilador puede inferir por si solo. Pero a veces, el compilador desconoce el tipo de valor que debe devolver una expresión como **`read "5"`**, que podría ser **`Int`**, **`Double`**, etc. Para saberlo, Haskell debe en realidad evaluar **`read "5"`**. Pero como Haskell es un lenguaje con tipos estáticos, debe conocer todos los tipos antes de que el código sea compilado (o en GHCi, evaluado). Así que con esto le estamos diciendo a Haskell: “Ey, esta expresión debe ser de este tipo en caso de que no sepas cual es”.

- Los miembros de la clase `Enum` son tipos secuencialmente ordenados, es decir, pueden ser enumerados. La principal ventaja de la clase de tipos **`Enum`** es que podemos usar los miembros en las listas aritméticas. También tienen definidos los sucesores y predecesores, por lo que podemos usar las funciones **`succ`** y **`pred`**. Los tipos de esta clase son: **`()`**, **`Bool`**, **`Char`**, **`Ordering`**, **`Int`**, **`Integer`**, **`Float`** y **`Double`**.

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

- Los miembros de `Bounded` poseen límites inferiores y superiores, es decir están acotados.

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

`minBound` y **`maxBound`** son interesantes ya que tienen el tipo **`(Bounded a) => a`**. Es decir, son constantes polimórficas.

Todas las tuplas son también **`Bounded`** si sus componentes los son también.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

- `Num` es la clase de tipos numéricos. Sus miembros tienen la propiedad de poder comportarse como números. Vamos a examinar el tipo de un número.

```
ghci> :t 20
20 :: (Num t) => t
```

Parece que todos los números son también constantes polimórficas. Pueden actuar como si fueran cualquier tipo de la clase **Num**.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Estos son los tipo estándar de la clase **Num**. Si examinamos el tipo de ***** veremos que puede aceptar cualquier tipo de número.

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Toma dos números del mismo tipo y devuelve un número del mismo tipo. Esa es la razón por la que **(5 :: Int) * (6 :: Integer)** lanzará un error mientras que **5 * (6 :: Integer)** funcionará correctamente y producirá un **Interger**, ya que **5** puede actuar como un **Integer** o un **Int**.

Para unirse a **Num**, un tipo debe ser amigo de **Show** y **Eq**.

- `Integral` es también un clase de tipos numérica. **Num** incluye todos los números, incluyendo números reales y enteros. **Integral** únicamente incluye números enteros. **Int** e **Integer** son miembros de esta clase.

- `Floating` incluye únicamente números en coma flotante, es decir **Float** y **Double**.

Una función muy útil para trabajar con números es `fromIntegral`. Tiene el tipo **fromIntegral :: (Num b, Integral a) => a -> b**. A partir de esta declaración podemos decir que toma un número entero y lo convierte en un número más general. Esto es útil cuando estas trabajando con números reales y enteros al mismo tiempo. Por ejemplo, la función **length** tiene el tipo **length :: [a] -> Int** en vez de tener un tipo más general como **(Num b) => length :: [a] -> b**. Creo que es por razones históricas o algo parecido, en mi opinión, es absurdo. De cualquier modo, si queremos obtener el tamaño de una lista y sumarle **3.2**, obtendremos un error al intentar sumar un entero con uno en coma flotante. Para solucionar esto, hacemos **fromIntegral (length [1,2,3,4]) + 3.2**.

Fíjate que en la declaración de tipo de **fromIntegral** hay varias restricciones de clase. Es completamente válido como puedes ver, las restricciones de clase deben ir separadas por comas y entre paréntesis.

La sintaxis de las funciones

Ajuste de patrones

En este capítulo cubriremos algunas de las construcciones sintácticas de Haskell más interesantes, empezando con el **ajuste de patrones** (“*pattern matching*” en inglés). Un ajuste de patrones consiste en una especificación de pautas que deben ser seguidas por los datos, los cuales pueden ser deconstruidos permitiéndonos acceder a sus componentes.

Podemos separar el cuerpo que define el comportamiento de una función en varias partes, de forma que el código quede mucho más elegante, limpio y fácil de leer. Podemos usar el ajuste de patrones con cualquier tipo de dato: números, caracteres, listas, tuplas, etc. Vamos a crear una función muy trivial que compruebe si el número que le pasamos es un siete o no.

```
lucky :: (Integral a) => a -> String
lucky 7 = "¡El siete de la suerte!"
lucky x = "Lo siento, ¡no es tu día de suerte!"
```

Cuando llamamos a **`lucky`**, los patrones son verificados de arriba a abajo y cuando un patrón concuerda con el valor asociado, se utiliza el cuerpo de la función asociado. En este caso, la única forma de que un número concuerde con el primer patrón es que dicho número sea 7. Si no lo es, se evaluara el siguiente patrón, el cual coincide con cualquier valor y lo liga a **`x`**. También se podría haber implementado utilizando una sentencia **`if`**. Pero, ¿qué pasaría si quisiéramos una función que nombrara los número del 1 al 5, o **`"No entre uno 1 y 5"`** para cualquier otro número? Si no tuviéramos el ajuste de patrones deberíamos crear un enrevesado árbol **`if then else`**. Sin embargo con él:

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "¡Uno!"
sayMe 2 = "¡Dos!"
sayMe 3 = "¡Tres!"
sayMe 4 = "¡Cuatro!"
sayMe 5 = "¡Cinco!"
sayMe x = "No entre uno 1 y 5"
```

Ten en cuenta que si movemos el último patrón (el más general) al inicio, siempre obtendríamos **`"No entre uno 1 y 5"`** como respuesta, ya que el primer patrón encajaría con cualquier número y no habría posibilidad de que se comprobaran los demás patrones.

¿Recuerdas la función factorial que creamos anteriormente? Definimos el factorial de un número **`n`** como **`product [1..n]`**. También podemos implementar una función factorial recursiva, de forma parecida a como lo haríamos en matemáticas. Empezamos diciendo que el factorial de 0 es 1. Luego decimos que el factorial de cualquier otro número entero positivo es ese entero multiplicado por el factorial de su predecesor.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Esta es la primera vez que definimos una función recursiva. La recursividad es muy importante en Haskell, pero hablaremos de ello más adelante. Resumiendo, esto es lo que pasa cuando intentamos obtener el factorial de, digamos 3. Primero intenta calcular **`3 * factorial 2`**. El factorial de 2 es **`2 * factorial 1`**, así que ahora tenemos **`3 * (2 * factorial 1)`**. **`factorial 1`** es **`1 * factorial 0`**, lo que nos lleva a **`3 * (2 * (1 * factorial 0))`**. Ahora viene el truco, hemos definido el factorial de 0 para que sea simplemente 1, y como se encuentra con ese patrón antes que el otro más general obtenemos 1. Así que el resultado equivale a **`3 * (2 * (1 * 1))`**. Si hubiésemos escrito el segundo patrón al inicio, hubiese aceptado todos los números incluyendo el 0 y el cálculo nunca terminaría. Por este motivo el orden es importante a la hora de definir los patrones y siempre es mejor definir los patrones más específicos al principio dejando los más generales al final.

Los patrones también pueden fallar. Si definimos una función como esta:

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

E intentamos ejecutarla con un valor no esperado, esto es lo que pasa:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function charName
```

Se queja porque tenemos un ajuste de patrones no exhaustivo y ciertamente así es. Cuando utilizamos patrones siempre tenemos que incluir uno general para asegurarnos que nuestro programa no fallará.

El ajuste de patrones también pueden ser usado con tuplas. ¿Cómo crearíamos una función que tomara dos vectores 2D (representados con duplas) y que devolviera la suma de ambos? Para sumar dos vectores sumamos primero sus componentes **`x`** y sus componentes **`y`** de forma separada. Así es como lo haríamos si no existiese el ajuste de patrones:

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Bien, funciona, pero hay mejores formas de hacerlo. Vamos a modificar la función para que utilice un ajuste de patrones.

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

¡Ahí lo tienes! Mucho mejor. Ten en cuenta que es un patrón general, es decir, se verificará para cualquier dupla. El tipo de **`addVectors`** es en ambos casos el mismo: **`addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)`**, por lo que está garantizado que tendremos dos duplas como parámetros.

`fst` y **`snd`** extraen componentes de las duplas. Pero, ¿qué pasa con las triplas? Bien, como no tenemos funciones que hagan lo mismo con las triplas vamos a crearlas nosotros mismos.

```
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y
```

```
third :: (a, b, c) -> c
third (_, _, z) = z
```

_ tiene el mismo significado que con las listas intensionales. Denota que en realidad no nos importa ese valor, ya que no lo vamos a utilizar.

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

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

En caso de que se produzca un fallo en el patrón, simplemente pasará al siguiente elemento.

Las listas también pueden ser usadas en un ajuste de patrones. Puedes comparar contra la lista vacía **[]** o contra cualquier patrón que involucre a **:** y la lista vacía. Como **[1,2,3]**, que solo es otra forma de expresar **1:2:3:[]** (podemos utilizar ambas alternativas). Un patrón como **x:xs** ligará la cabeza de la lista con **x** y el resto con **xs**, incluso cuando la lista tenga solo un elemento, en cuyo caso **xs** acabará siendo la lista vacía.

Nota
El patrón **x:xs** es muy utilizado, especialmente con las funciones recursivas. Los patrones que contengan un **:** solo aceptarán listas con algún elemento.

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

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

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

Comprobamos que funciona:

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

¡Bien! Fíjate que si queremos ligar varias variables (incluso aunque alguna de ellas sea **_** y realmente no la queremos ligar) debemos rodearlas con paréntesis. Fíjate también en la función **error** que acabamos de utilizar. Ésta toma una cadena y genera un error en tiempo de ejecución usado la cadena que le pasemos como información acerca del error que ocurrió. Provoca que el programa termine, lo cual no es bueno usar muy a menudo. De todas formas, llamar a **head** con una lista vacía no tiene mucho sentido.

Vamos a crear una función que nos diga algunos de los primeros elementos que contiene una lista.

```
tell :: (Show a) => [a] -> String
tell []       = "La lista está vacía"
tell (x:[])   = "La lista tiene un elemento: " ++ show x
tell (x:y:[]) = "La lista tiene dos elementos: " ++ show x ++ " y " ++ show y
tell (x:y:_)  = "La lista es larga. Los primeros dos elementos son: " ++ show x ++ " y "
```

Esta función es segura ya que tiene en cuenta la posibilidad de una lista vacía, una lista con un elemento, una lista con dos elementos y una lista con más de dos elementos. Date cuenta que podríamos escribir **`(x:[])`** y **`(x:y:[])`** como **`[x]`** y **`[x,y]`** sin usar paréntesis. Pero no podemos escribir **`(x:y:_)`** usando corchetes ya que acepta listas con más de dos elementos.

Ya implementamos la función **`length`** usando listas intensionales. Ahora vamos a implementarla con una pizca de recursión.

```
length' :: (Num b) => [a] -> b
length' [] = 0
length' (_:xs) = 1 + length' xs
```

Es similar a la función factorial que escribimos antes. Primero definimos el resultado de una entrada conocida, la lista vacía. Esto también es conocido como el caso base. Luego en el segundo patrón dividimos la lista en su cabeza y el resto. Decimos que la longitud es 1 más el tamaño del resto de la lista. Usamos **`_`** para la cabeza de la lista ya que realmente no nos interesa su contenido. Fíjate que también hemos tenido en cuenta todos los posibles casos de listas. El primer patrón acepta la lista vacía, y el segundo todas las demás.

Vamos a ver que pasa si llamamos a **`length'`** con **`"ojo"`**. Primero se comprobaría si es una lista vacía, como no lo es continuaríamos al siguiente patrón. Éste es aceptado y nos dice que la longitud es **`1 + length' "jo"`**, ya que hemos divido la cadena en cabeza y cola, decapitando la lista. Vale. El tamaño de **`"jo"`** es, de forma similar, **`1 + length' "o"`**. Así que ahora mismo tenemos **`1 + (1 + length' "o")`**. **`length' "o"`** es **`1 + length' ""`** (también lo podríamos escribir como **`1 + length' []`**). Y como tenemos definido **`length' []`** a 0, al final tenemos **`1 + (1 + (1 + 0))`**.

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

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

También existen los llamados *patrones como*, o *patrones as* (del inglés, *as patterns*). Son útiles para descomponer algo usando un patrón, de forma que se ligue con las variables que queramos y además podamos mantener una referencia a ese algo como un todo. Para ello ponemos un **`@`** delante del patrón. La mejor forma de entenderlo es con un ejemplo: **`xs@(x:y:ys)`**. Este patrón se ajustará exactamente a lo mismo que lo haría **`x:y:ys`** pero además podríamos acceder fácilmente a la lista completa usando **`xs`** en lugar de tener que repetirnos escribiendo **`x:y:ys`** en el cuerpo de la función. Un ejemplo rápido:

```
capital :: String -> String
capital "" = "¡Una cadena vacía!"
capital all@(x:_) = "La primera letra de " ++ all ++ " es " ++ [x]
```

```
ghci> capital "Dracula"
"La primera letra de Dracula es D"
```

Normalmente usamos los *patrones como* para evitar repetirnos cuando estamos ajustando un patrón más grande y tenemos que usarlo entero otra vez en algún lugar del cuerpo de la función.

Una cosa más, no podemos usar **++** en los ajustes de patrones. Si intentamos usar un patrón **(xs ++ ys)**, ¿qué habría en la primera lista y qué en la segunda? No tiene mucho sentido. Tendría más sentido ajustar patrones como **(xs ++ [x,y,z])** o simplemente **(xs ++ [x])** pero dada la naturaleza de las listas no podemos hacer esto.

¡Guardas, Guardas!

Mientras que los patrones son una forma de asegurarnos que un valor tiene una determinada forma y deconstruirlo, las guardas son una forma de comprobar si alguna propiedad de una valor (o varios de ellos) es cierta o falsa. Suena muy parecido a una sentencia **if** y de hecho es muy similar. La cuestión es que las guardas son mucho más legibles cuando tienes varias condiciones y encajan muy bien con los patrones.

En lugar de explicar su sintaxis, simplemente vamos a crear una función que utilice guardas. Crearemos una función simple que te regañará de forma diferente en función de tu IMC (índice de masa corporal). Tu IMC es igual a tu altura dividida por tu peso al cuadrado. Si tu IMC es menor que 18,5 tienes infrapeso. Si estas en algún lugar entre 18,5 y 25 eres del montón. Si tienes entre 25 y 30 tienes sobrepeso y si tienes más de 30 eres obeso. Así que aquí tienes la función (no estamos calculando nada ahora, simplemente obtiene un IMC y te regaña)

```
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
    | bmi <= 18.5 = "Tienes infrapeso ¿Eres emo?"
    | bmi <= 25.0 = "Supuestamente eres normal... Espero que seas feo."
    | bmi <= 30.0 = "¡Estás gordo! Pierde algo de peso gordito."
    | otherwise   = "¡Enhorabuena, eres una ballena!"
```

Las guardas se indican con barras verticales que siguen al nombre de la función y sus parámetros. Normalmente tienen una sangría y están alineadas. Una guarda es básicamente una expresión booleana. Si se evalúa a **True**, entonces el cuerpo de la función correspondiente es utilizado. Si se evalúa a **False**, se comprueba la siguiente guarda y así sucesivamente. Si llamamos a esta función con **24.3**, primero comprobará si es menor o igual que **18.5**. Como no lo es, seguirá a la siguiente guarda. Se comprueba la segunda guarda y como 24,3 es menor que 25, se devuelve la segunda cadena.

Recuerda a un gran árbol **if then else** de los lenguajes imperativos, solo que mucho más claro. Generalmente los arboles **if else** muy grandes están mal vistos, pero hay ocasiones en que un problema se define de forma discreta y no hay forma de solucionarlo. Las guardas son una buena alternativa para esto.

Muchas veces la última guarda es **otherwise**. **otherwise** está definido simplemente como **otherwise = True** y acepta todo. Es muy similar al ajuste de patrones, solo se aceptan si la entrada satisface un patrón, pero las guardas comprueban condiciones booleanas. Si todas las guardas de una función se evalúan a **False** (y no hemos dado otra guarda **otherwise**), la evaluación falla y continuará hacia el siguiente **patrón**. Por esta razón los patrones y las guardas encajan tan bien juntas. Si no existe ningún patrón ni ninguna guarda aceptable se lanzará un error.

Por supuesto podemos usar guardas con funciones que tomen tantos parámetros como se quieran. En lugar de dejar que el usuario tenga que calcular su propio IMC por su cuenta antes de llamar a la función, vamos a modificar la función para que tome la altura y el peso y lo calcule por nosotros.

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "Tienes infrapeso ¿Eres emo?"
    | weight / height ^ 2 <= 25.0 = "Supuestamente eres normal... Espero que seas feo."
    | weight / height ^ 2 <= 30.0 = "¡Estás gordo! Pierde algo de peso gordito."
    | otherwise                   = "¡Enhorabuena, eres una ballena!"
```

Vamos a ver si estoy gordo...

```
ghci> bmiTell 85 1.90
"Supuestamente eres normal... Espero que seas feo."
```

¡Sí! No estoy gordo, pero Haskell me acaba de llamar feo...

Fíjate que no hay un **=** después del nombre de la función y sus parámetros, antes de la primera guarda. Muchos novatos obtienen un error sintáctico por poner un **=** ahí, y tú también lo harás.

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

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

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

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

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

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


ghci> 3 `myCompare` 2
GT
```

Nota

No solo podemos llamar a funciones de forma infija usando las comillas, sino que también podemos definirlas de esta forma. A veces es más fácil leerlo así.

¿Dónde?

En la sección anterior definimos la función que calculaba el IMC así:

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "Tienes infrapeso ¿Eres emo?"
    | weight / height ^ 2 <= 25.0 = "Supuestamente eres normal... Espero que seas feo."
    | weight / height ^ 2 <= 30.0 = "¡Estás gordo! Pierde algo de peso gordito."
    | otherwise                   = "¡Enhorabuena, eres una ballena!"
```

Si te fijas notarás que nos repetimos tres veces. Nos repetimos tres veces. Repetirse (tres veces) mientras estas programando es tan deseable como que te den una patada donde más te duela. Ya que estamos repitiendo la misma expresión tres veces sería ideal si pudiésemos calcularla una sola vez, ligarla a una variable y utilizarla en lugar de la expresión. Bien, podemos modificar nuestra función de esta forma:

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "Tienes infrapeso ¿Eres emo?"
    | bmi <= 25.0 = "Supuestamente eres normal... Espero que seas feo."
    | bmi <= 30.0 = "¡Estás gordo! Pierde algo de peso gordito."
    | otherwise   = "¡Enhorabuena, eres una ballena!"
    where bmi = weight / height ^ 2
```

Hemos puesto la palabra reservada **where** después de las guardas (normalmente es mejor alinearla con el resto de las barras verticales) y luego definimos varias variables. Estas variables son visibles en las guardas y nos dan la ventaja de no tener que repetirnos. Si decidimos que tenemos que calcular el IMC de otra forma solo tenemos que modificarlo en un lugar. También mejora la legibilidad ya que da nombre a las cosas y hace que nuestros programas sean más rápidos ya que cosas como **bmi** solo deben calcularse una vez. Podríamos pasarnos un poco y presentar una función como esta:

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "Tienes infrapeso ¿Eres emo?"
    | bmi <= normal = "Supuestamente eres normal... Espero que seas feo."
    | bmi <= fat    = "¡Estás gordo! Pierde algo de peso gordito."
    | otherwise     = "¡Enhorabuena, eres una ballena!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Las variables que definamos en la sección **where** de una función son solo visibles desde esa función, así que no nos tenemos que preocupar de ellas a la hora de crear más variables en otras funciones. Si no alineamos la sección **where** bien y de forma correcta, Haskell se confundirá porque no sabrá a que grupo pertenece.

Las variables definidas con **where** no se comparten entre los cuerpos de diferentes patrones de una función. Si queremos que varios patrones accedan a la misma variable debemos definirla de forma global.

También podemos usar el ajuste de patrones con las secciones **where**. Podríamos reescribir la sección **where** de nuestra función anterior como:

```haskell
...
where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Vamos a crear otra función trivial en el que dado un nombre y un apellido devuelva sus iniciales.

```haskell
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Podríamos haber realizado el ajuste de patrones directamente en los parámetros de la función (en realidad hubiese sido más corto y elegante) pero así podemos ver lo que es posible hacer con las secciones **where**.

De la misma forma que hemos definido constantes en los bloques **where** también podemos definir funciones. Manteniéndonos fieles a nuestro programa de salud vamos a hacer una función que tome una lista de duplas de pesos y estaturas y devuelva una lista de IMCs.

```haskell
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
```

```
    where bmi weight height = weight / height ^ 2
```

¡Ahí lo tienes! La razón por la que hemos creado la función **bmi** en este ejemplo es que no podemos calcular simplemente un IMC desde los parámetros de nuestra función. Tenemos que examinar todos los elementos de la lista y calcular su IMC para cada dupla.

Las secciones **where** también pueden estar anidadas. Es muy común crear una función y definir algunas funciones auxiliares en la sección **where** y luego definir otras funciones auxiliares dentro de cada uno de ellas.

Let it be

Muy similar a las secciones **where** son las expresiones **let**. Las secciones **where** son una construcción sintáctica que te dejan ligar variables al final de una función de forma que toda la función pueda acceder a ella, incluyendo todas las guardas. Las expresiones **let** sirven para ligar variables en cualquier lugar y son expresiones en si mismas, pero son muy locales, así que no pueden extenderse entre las guardas. Tal y como todas las construcciones de Haskell que te permiten ligar valores a variables, las expresiones **let** permiten usar el ajuste de patrones. ¡Vamos a verlo en acción! Así es como podríamos definir una función que nos diera el área de un cilindro basándose en su altura y su radio.

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^2
    in  sideArea + 2 * topArea
```

Su forma es **let <definición> in <expresión>**. Las variables que definamos en la expresión **let** son accesibles en la parte **in**. Como podemos ver, también podríamos haber definido esto con una sección **where**. Fíjate también que los nombres están alineados en la misma columna. Así que, ¿cuál es la diferencia entre ellos? Por ahora parece que **let** pone las definiciones primero y luego la expresión que las utiliza mientras que **where** lo hace en el orden inverso.

La diferencia es que las expresiones **let** son expresiones por si mismas. Las secciones **where** son simplemente construcciones sintácticas. ¿Recuerdas cuando explicamos las sentencias **if** y se explicó que como son una expresión pueden ser usadas en casi cualquier lugar?

```
ghci> [if 5 > 3 then "Woo" else "Boo", if 'a' > 'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

También puedes hacer lo mismo con las expresiones **let**.

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

También pueden ser utilizadas para definir funciones en un ámbito local:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

Si queremos ligar varias variables en una solo línea, obviamente no podemos alinear las definiciones en la misma columna. Por este motivo podemos separarlas con puntos y comas.

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ b
(6000000,"Hey there!")
```

No tenemos porque poner el último punto y coma pero podemos hacerlo si queremos. Como ya hemos dicho, podemos utilizar ajustes de patrones con las expresiones **`let`**. Son muy útiles para desmantelar tuplas en sus componentes y ligarlos a varias variables.

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

También podemos usar las secciones **`let`** dentro de las listas intensionales. Vamos a reescribir nuestro ejemplo anterior que calculaba una lista de duplas de alturas y pesos para que use un **`let`** dentro de una lista intensional en lugar de definir una función auxiliar con un **`where`**.

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

Incluimos un **`let`** dentro de la lista intensional como si fuera un predicado, solo que no filtra los elementos, únicamente liga variables. Las variables definidas en una expresión **`let`** dentro de una lista intensional son visibles desde la función de salida (la parte anterior a **|**) y todos los predicados y secciones que vienen después de su definición. Podríamos hacer que nuestra función devolviera el IMC solo para la gente obesa así:

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

No podemos usar el nombre **`bmi`** dentro de la parte **`(w, h) <- xs`** ya que está definida antes que la expresión **`let`**.

Omitimos la parte **`in`** de las secciones **`let`** dentro de las lista intensionales porque la visibilidad de los nombres está predefinida en estos casos. Sin embargo, podemos usar una sección **`let in`** en un predicado y las variables definidas solo serán visibles en este predicado. La parte **`in`** también puede ser omitida cuando definimos funciones y constantes dentro del intérprete **`GHCi`**. Si lo hacemos, las variables serán visibles durante toda la sesión.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Si las expresiones **`let`** son tan interesantes, ¿por qué no usarlas siempre en lugar de las secciones **`where`**? Bueno, como las expresiones **`let`** son expresiones y son bastante locales en su ámbito, no pueden ser usadas entre guardas. Hay gente que prefiere las secciones **`where`** porque las variables vienen después de la función que los utiliza. De esta forma, el cuerpo de la función esta más cerca de su nombre y declaración de tipo y algunos piensan que es más legible.

Expresiones case

Muchos lenguajes imperativos (como C, C++, Java, etc.) tienen construcciones sintácticas

case y si alguna vez has programado en ellos, probablemente sepas acerca de que va esto. Se trata de tomar una variable y luego ejecutar bloques de código para ciertos valores específicos de esa variable y luego incluir quizá algún bloque que siempre se ejecute en caso de que la variable tenga algún valor que no se ajuste con ninguno de los anteriores.

Haskell toma este concepto y lo lleva un paso más allá. Como su nombre indica las expresiones **case** son, bueno, expresiones, como las expresiones **if else** o las expresiones **let**. No solo podemos evaluar expresiones basándonos en los posibles valores de un variable sino que podemos realizar un ajuste de patrones. Mmmm... tomar un valor, realizar un ajuste de patrones sobre él, evaluar trozos de código basados en su valor, ¿dónde hemos oído esto antes? Oh sí, en los ajuste de patrones de los parámetros de una función. Bueno, en realidad es una alternativa sintáctica para las expresiones **case**. Estos dos trozos de código hacen lo mismo y son intercambiables:

```
head' :: [a] -> a
head' [] = error "¡head no funciona con listas vacías!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "¡head no funciona con listas vacías!"
                      (x:_) -> x
```

Como puedes ver la sintaxis para las expresiones **case** es muy simple.

```
case expresion of patron -> resultado
                  patron -> resultado
                  patron -> resultado
                  ...
```

La expresión es ajustada contra los patrones. La acción de ajuste de patrones se comporta como se espera: el primer patrón que se ajuste es el que se utiliza. Si no se puede ajustar a ningún patrón de la expresión **case** se lanzará un error de ejecución.

Mientras que el ajuste de patrones de los parámetros de una función puede ser realizado únicamente al definir una función, las expresiones **case** pueden ser utilizadas casi en cualquier lugar. Por ejemplo:

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

Son útiles para realizar un ajuste de patrones en medio de una expresión. Como el ajuste de patrones que se realiza en la definición de una función es una alternativa sintáctica a las expresiones **case**, también podríamos utilizar algo como esto:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

Recursión

¡Hola recursión!

En el capítulo anterior ya mencionamos la recursión. En este capítulo veremos más detenidamente este tema, el porqué es importante en Haskell y como podemos crear soluciones a problemas de forma elegante y concisa.

Si aún no sabes que es la recursión, lee esta frase: La recursión es en realidad una forma de definir funciones en la que dicha función es utiliza en la propia definición de la función. Las definiciones matemáticas normalmente están definidas de forma recursiva. Por ejemplo, la serie de Fibonacci se define recursivamente. Primero, definimos los dos primeros números de Fibonacci de forma no recursiva. Decimos que **`F(0) = 0`** y **`F(1) = 1`**, que significa que el 1º y el 2º número de Fibonacci es 0 y 1, respectivamente. Luego, para cualquier otro índice, el número de Fibonacci es la suma de los dos números de Fibonacci anteriores. Así que **`F(n) = F(n-1) + F(n-2)`**. De esta forma, **`F(3) = F(2) + F(1)`** que es **`F(3) = (F(1) + F(0)) + F(1)`**. Como hemos bajado hasta los únicos números definidos no recursivamente de la serie de Fibonacci, podemos asegurar que **`F(3) = 2`**. Los elementos definidos no recursivamente, como **`F(0)`** o **`F(1)`**, se llaman **casos base**, y si tenemos solo casos base en una definición como en **`F(3) = (F(1) + F(0)) + F(1)`** se denomina **condición límite**, la cual es muy importante si quieres que tu función termine. Si no hubiéramos definido **`F(0)`** y **`F(1)`** no recursivamente, nunca obtendríamos un resultado para un número cualquiera, ya que alcanzaríamos **`0`** y continuaríamos con los número negativos. De repente, encontraríamos un **`F(-2000) = F(-2001) + F(-2002)`** y seguiríamos sin ver el final.

La recursión es muy importante en Haskell ya que, al contrario que en los lenguajes imperativos, realizamos cálculos declarando como **es** algo, en lugar de declarar **como** obtener algo. Por este motivo no hay bucles **`while`** o bucles **`for`** en Haskell y en su lugar tenemos que usar la recursión para declarar como es algo.

El impresionante maximum

La función **`maximum`** toma una lista de cosas que pueden ser ordenadas (es decir instancias de la clase de tipos **`Ord`**) y devuelve la más grande. Piensa en como implementaríamos esto de forma imperativa. Probablemente crearíamos una variable para mantener el valor máximo hasta el momento y luego recorreríamos los elementos de la lista de forma que si un elemento es mayor que el valor máximo actual, lo remplazaríamos. El máximo valor que se mantenga al final es el resultado. ¡Wau! son muchas palabras para definir un algoritmo tan simple.

Ahora vamos a ver como definiríamos esto de forma recursiva. Primero podríamos establecer un caso base diciendo que el máximo de una lista unitaria es el único elemento que contiene la lista. Luego podríamos decir que el máximo de una lista más larga es la cabeza de esa lista si es mayor que el máximo de la cola, o el máximo de la cola en caso de que no lo sea. ¡Eso es! Vamos a implementarlo en Haskell.

```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "Máximo de una lista vacía"
maximum' [x] = x
maximum' (x:xs)
```

```
    | x > maxTail = x
    | otherwise   = maxTail
  where maxTail = maximum' xs
```

Como puedes ver el ajuste de patrones funcionan genial junto con la recursión. Muchos lenguajes imperativos no tienen patrones así que hay que utilizar muchos **`if`**/**`else`** para implementar los casos base. El primer caso base dice que si una lista está vacía, ¡Error! Tiene sentido porque, ¿cuál es el máximo de una lista vacía? Ni idea. El segundo patrón también representa un caso base. Dice que si nos dan una lista unitaria simplemente devolvemos el único elemento.

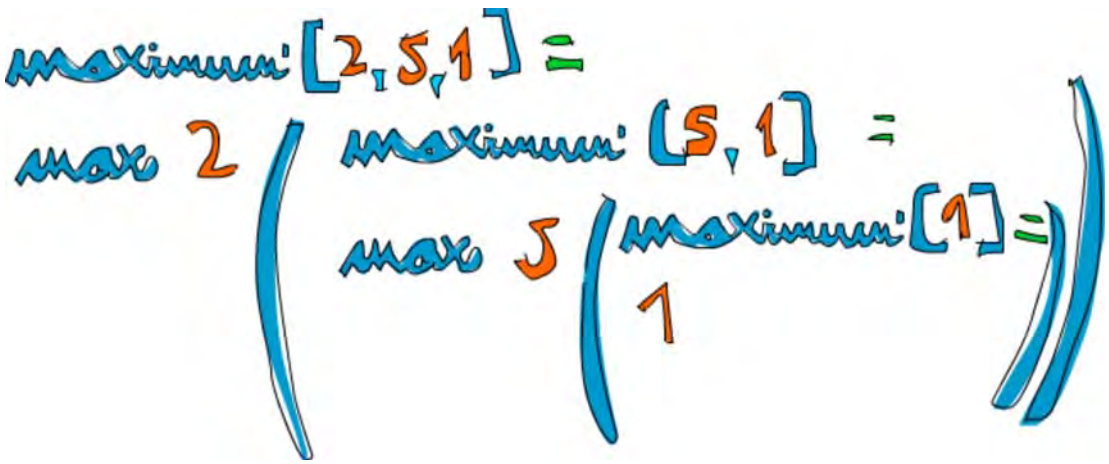
En el tercer patrón es donde está la acción. Usamos un patrón para dividir la lista en cabeza y cola. Esto es algo muy común cuando usamos una recursión con listas, así que ve acostumbrándote. Usamos una sección **`where`** para definir **`maxTail`** como el máximo del resto de la lista. Luego comprobamos si la cabeza es mayor que el resto de la cola. Si lo es, devolvemos la cabeza, si no, el máximo del resto de la lista.

Vamos a tomar una lista de números de ejemplo y comprobar como funcionaria: **`[2,5,1]`**. Si llamamos **`maximum'`** con esta lista, los primeros dos patrones no ajustarían. El tercero si lo haría y la lista se dividiría en **`2`** y **`[5,1]`**. La sección **`where`** requiere saber el máximo de **`[5,1]`** así que nos vamos por ahí. Se ajustaría con el tercer patrón otra vez y **`[5,1]`** sería dividido en **`5`** y **`[1]`**. Otra vez, la sección **`where`** requiere saber el máximo de **`[1]`**. Como esto es un caso base, devuelve 1 ¡Por fin! Así que subimos un paso, comparamos **`5`** con el máximo de **`[1]`** (que es **`1`**) y sorprendentemente obtenemos 5. Así que ahora sabemos que el máximo de **`[5,1]`** es **`5`**. Subimos otro paso y tenemos **`2`** y **`[5,1]`**. Comparamos **`2`** con el máximo de **`[5,1]`**, que es **`5`** y elegimos **`5`**.

Una forma más clara de escribir la función **`maximum'`** es usando la función **`max`**. Si recuerdas, la función **`max`** toma dos cosas que puedan ser ordenadas y devuelve la mayor de ellas. Así es como podríamos reescribir la función utilizando **`max`**:

```
maximum' :: (Ord a) => [a] -> a
maximum' []     = error "maximum of empty list"
maximum' [x]    = x
maximum' (x:xs) = x `max` (maximum' xs)
```

¿A que es elegante? Resumiendo, el máximo de una lista es el máximo entre su primer elemento y el máximo del resto de sus elementos.



Unas cuantas funciones recursivas más

Ahora que sabemos cómo pensar de forma recursiva en general, vamos a implementar unas cuantas funciones de forma recursiva. En primer lugar, vamos a implementar **`replicate`**. **`replicate`** toma un **`Int`** y algún elemento y devuelve una lista que contiene varias repeticiones de ese mismo elemento. Por ejemplo, **`replicate 3 5`** devuelve **`[5,5,5]`**. Vamos a pensar en el caso base. Mi intuición me dice que el caso base es 0 o menos. Si intentamos replicar algo 0 o menos veces, debemos devolver

una lista vacía. También para números negativos ya que no tiene sentido.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

Aquí usamos guardas en lugar de patrones porque estamos comprobando una condición booleana. Si **n** es menor o igual que 0 devolvemos una lista vacía. En otro caso devolvemos una lista que tiene **x** como primer elemento y **x** replicado n-1 veces como su cola. Finalmente, la parte n-1 hará que nuestra función alcance el caso base.

Ahora vamos a implementar **take**. Esta función toma un cierto número de elementos de una lista. Por ejemplo, **take 3 [5,4,3,2,1]** devolverá **[5,4,3]**. Si intentamos obtener 0 o menos elementos de una lista, obtendremos una lista vacía. También si intentamos tomar algo de una lista vacía, obtendremos una lista vacía. Fíjate que ambos son casos base. Vamos a escribirlo.

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

El primer patrón indica que si queremos obtener 0 o un número negativo de elementos, obtenemos una lista vacía. Fíjate que estamos usando **_** para enlazar la lista ya que realmente no nos importa en este patrón. Además también estamos usando una guarda, pero sin la parte **otherwise**. Esto significa que si **n** acaba siendo algo más que 0, el patrón fallará y continuará hacia el siguiente. El segundo patrón indica que si intentamos tomar algo de una lista vacía, obtenemos una lista vacía. El tercer patrón rompe la lista en cabeza y cola. Luego decimos que si tomamos **n** elementos de una lista es igual a una lista que tiene **x** como cabeza y como cola una lista que tome n-1 elementos de la cola. Intenta usar papel y lápiz para seguir el desarrollo de como sería la evaluación de **take 3 [4,3,2,1]**, por ejemplo.

reverse simplemente pone al revés una lista. Piensa en el caso base, ¿cuál es? Veamos... ¡Es una lista vacía! Una lista vacía inversa es igual a esa misma lista vacía. Vale, ¿qué hay del resto de la lista? Podríamos decir que si dividimos una lista en su cabeza y cola, la lista inversa es igual a la cola invertida más luego la cabeza al final.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

¡Ahí lo tienes!

Como Haskell soporta listas infinitas, en realidad nuestra recursión no tiene porque tener casos base. Pero si no los tiene, seguiremos calculando algo infinitamente o bien produciendo una estructura infinita. Sin embargo, lo bueno de estas listas infinitas es que podemos cortarlas por donde queramos. **repeat** toma un elemento y devuelve una lista infinita que simplemente tiene ese elemento. Una implementación recursiva extremadamente simple es:

```
repeat' :: a -> [a]
repeat' x = x : repeat' x
```

Llamando a **`repeat 3`** nos daría una lista que tiene un **`3`** en su cabeza y luego tendría una lista infinita de treses en su cola. Así que **`repeat 3`** se evaluaría a algo como **`3:(repeat 3)`**, que es **`3:(3:(repeat 3))`**, que es **`3:(3:(3:(repeat 3)))`**, etc. **`repeat 3`** nunca terminará su evaluación, mientras que **`take 5 (repeat 3)`** nos devolverá un lista con cinco treses. Es igual que hacer **`replicate 5 3`**.

`zip` toma dos listas y las combina en una. **`zip [1,2,3] [2,3]`** devuelve **`[(1,2),(2,3)]`** ya que trunca la lista más larga para que coincida con la más corta. ¿Qué pasa si combinamos algo con la lista vacía? Bueno, obtendríamos un una lista vacía. Así que es este es nuestro caso base. Sin embargo, **`zip`** toma dos listas como parámetros, así que en realidad tenemos dos casos base.

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

Los dos primeros patrones dicen que si la primera o la segunda lista están vacías entonces obtenemos una lista vacía. Combinar **`[1,2,3]`** y **`['a','b']`** finalizará intentando combinar **`[3]`** y **`[]`**. El caso base aparecerá en escena y el resultado será **`(1,'a'):(2,'b'):[]`** que exactamente lo mismo que **`[(1,'a'),(2,'b')]`**.

Vamos a implementar una función más de la biblioteca estándar, **`elem`**, que toma un elemento y una lista y busca si dicho elemento está en esa lista. El caso base, como la mayoría de las veces con las listas, es la lista vacía. Sabemos que una lista vacía no contiene elementos, así que lo más seguro es que no contenga el elemento que estamos buscando...

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Bastante simple y previsible. Si la cabeza no es elemento que estamos buscando entonces buscamos en la cola. Si llegamos a una lista vacía, el resultado es falso.

¡Quicksort!

Tenemos una lista de elementos que pueden ser ordenados. Su tipo es miembro de la clase de tipos **`Ord`**. Y ahora, queremos ordenarlos. Existe un algoritmo muy interesante para ordenarlos llamado Quicksort. Es una forma muy inteligente de ordenar elementos. Mientras en algunos lenguajes imperativos puede tomar hasta 10 líneas de código para implementar Quicksort, en Haskell la implementación es mucho más corta y elegante. Quicksort se ha convertido en usa especie de pieza de muestra de Haskell. Por lo tanto, vamos a implementarlo, a pesar de que la implementación de Quicksort en Haskell se considera muy cursi ya que todo el mundo lo hace en las presentaciones para que veamos los bonito que es.

Bueno, la declaración de tipo será **`quicksort :: (Ord a) => [a] -> [a]`**. Ninguna sorpresa. ¿Caso base? La lista vacía, como era de esperar. Ahora viene el algoritmo principal: una lista ordenada es una lista que tiene todos los elementos menores (o iguales) que la cabeza al principio (y esos valores están ordenados), luego viene la cabeza de la lista que estará en el medio y luego vienen los elementos que son mayores que la cabeza (que también estarán ordenados). Hemos dicho dos veces “ordenados”, así que probablemente tendremos que hacer

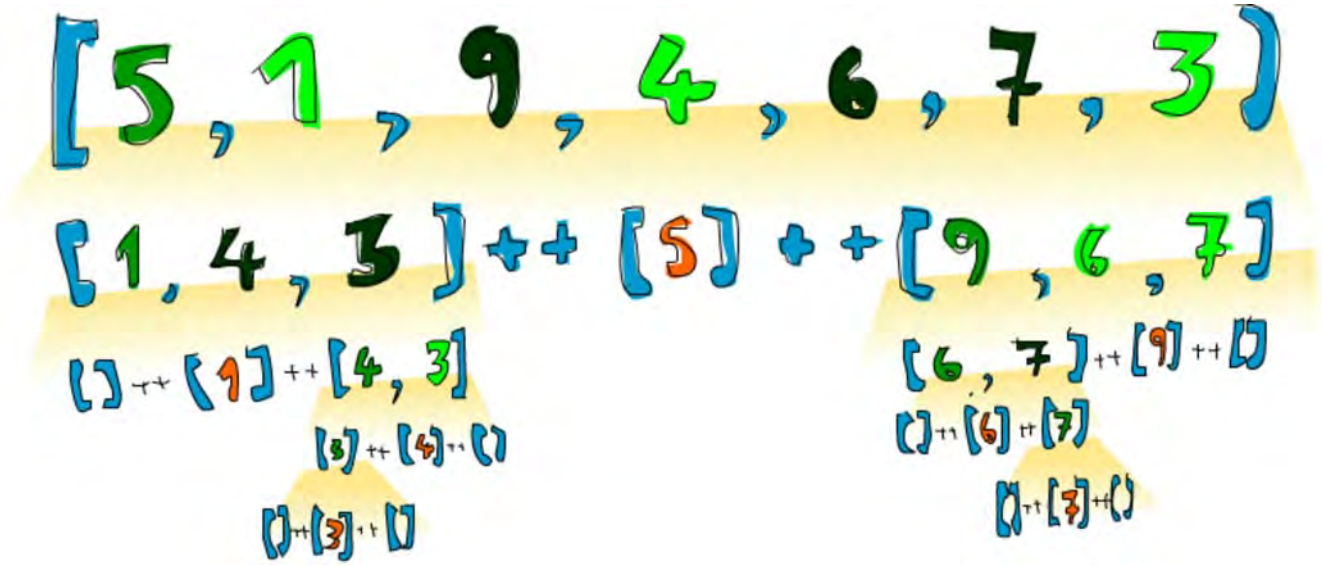
dos llamadas recursivas. También hemos usado dos veces el verbo “es” para definir el algoritmo en lugar de “hace esto”, “hace aquello”, “entonces hace”... ¡Esa es la belleza de la programación funcional! ¿Cómo vamos a conseguir filtrar los elementos que son mayores y menores que la cabeza de la lista? Con listas intensionales. Así que empecemos y definamos esta función:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted  = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Vamos a ejecutar una pequeña prueba para ver si se comporta correctamente.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "el veloz murcielago hindu comia feliz cardillo y kiwi"
"        aaacccddeeeefghiiiiiiiklllllllmmnoooorruuvwyzz"
```

Bien ¡De esto estábamos hablando! Así que si tenemos, digamos **`[5,1,9,4,6,7,3]`** y queremos ordenarlos, el algoritmo primero tomará la cabeza de la lista, que es 5 y lo pondrá en medio de dos listas que son los menores y los mayores de este. De esta forma tendremos **`(quicksort [1,4,3]) ++ [5] ++ (quicksort [9,6,7])`**. Sabemos que cuando la lista este completamente ordenada, el número **`5`** permanecerá en la cuarta posición ya que hay tres números menores y tres números mayores que él. Ahora si ordenamos **`[1,4,3]`** y **`[9,6,7]`**, ¡tendremos una lista ordenada! Ordenamos estas dos listas utilizando la misma función. Al final llegaremos a un punto en el que alcanzaremos listas vacías y las listas vacías ya están ordenadas de alguna forma. Aquí tienes una ilustración:



Un elemento que está en su posición correcta y no se moverá más está en naranja. Leyendo de izquierda a derecha estos elemento la lista aparece ordenada. Aunque elegimos comparar todos los elementos con la cabeza, podríamos haber elegido cualquier otro elemento. En Quicksort, se llama pivote al elemento con el que comparamos. Estos son los de color verde. Elegimos la cabeza porque es muy fácil aplicarle un patrón. Los elementos que son más pequeños que el pivote son de color verde claro y los elementos que son mayores en negro. El gradiente amarillo representa la aplicación de Quicksort.

Pensando de forma recursiva

Hemos usado un poco la recursión y como te habrás dado cuenta existen unos pasos comunes. Normalmente primero definimos los casos base y luego definimos una función que hace algo entre un elemento y la función aplicada al resto de elementos. No importa si este elemento es una lista, un árbol o cualquier otra estructura de datos. Un sumatorio es la suma del

primer elemento más la suma del resto de elementos. Un productorio es el producto del primer elemento entre el producto del resto de elementos. El tamaño de una lista es 1 más el tamaño del resto de la lista, etc.

Por supuesto también existen los casos base. Por lo general un caso base es un escenario en el que la aplicación de una recursión no tiene sentido. Cuando trabajamos con listas, los casos base suelen tratar con listas vacías. Cuando utilizamos árboles los casos base son normalmente los nodos que no tienen hijos.

Es similar cuando tratamos con números. Normalmente hacemos algo con un número y luego aplicamos la función a ese número modificado. Ya hicimos funciones recursivas de este tipo como el del factorial de un número, el cual no tiene sentido con cero, ya que el factorial solo está definido para enteros positivos. A menudo el caso base resulta ser la identidad. La identidad de la multiplicación es 1 ya que si multiplicas algo por 1 obtienes el mismo resultado. También cuando realizamos sumatorios de listas, definimos como 0 al sumatorio de una lista vacía, ya que 0 es la identidad de la suma. En Quicksort, el caso base es la lista vacía y la identidad es también la lista vacía, ya que si añades a una lista la lista vacía obtienes la misma lista ordenada.

Cuando queremos resolver un problema de forma recursiva, primero pensamos donde no se aplica una solución recursiva y si podemos utilizar esto como un caso base. Luego pensamos en las identidades, por donde deberíamos romper los parámetros (por ejemplo, las lista se rompen en cabeza y cola) y en que parte deberíamos aplicar la función recursiva.

Funciones de orden superior

Las funciones de Haskell pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama función de orden superior. Las funciones de orden superior no son simplemente una parte más de Haskell, ellas mismas representan la experiencia de programar en Haskell. Aparecen cuando quieres definir cálculos definiendo cosas como son en lugar de definir los pasos de cambio de algún estado o algún bucle, las funciones de orden superior son indispensables. Son realmente una forma muy potente de resolver problemas y de pensar acerca de los programas.

Funciones currificadas

Oficialmente cada función de Haskell solo puede tomar un parámetro. Así que ¿Como es posible que hayamos definido y usado varias funciones que toman mas de un parámetro? Bueno ¡Es un buen truco! Todas las funciones que hemos usado hasta el momento y aceptaban más de un parámetro han sido funciones currificadas ¿Qué significa esto? Lo entenderás mejor con un ejemplo. Vamos a usar a nuestro buen amigo, la función **`max`**. Parece que toma dos parámetro y devuelve aquél que es mayor. Al aplicar **`max 4 5`** primero se crea una función que toma un solo parámetro y devuelve 4 o el parámetro, dependiendo de cual sea mayor. Luego, 5 es aplicado a esa función y esta produce el resultado deseado. Esto suena un poco complicado pero en realidad es un concepto muy útil. Las siguientes dos llamadas son equivalentes:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

El poner un espacio entre dos cosas es sencillamente **aplicar una función**. El espacio es una especie de operador y tiene el orden de preferencia mayor. Vamos a examinar el tipo de **`max`**. Es **`max :: (Ord a) => a -> a -> a`**. Esto también puede ser escrito como **`max :: (Ord a) => a -> (a -> a)`**. Y también puede leerse como: **`max`** toma un **`a`** y devuelve (eso es **`->`**) una función que toma un **`a`** y devuelve un **`a`**. Ese es el porqué el tipo devuelto y los parámetros de la función están separados solamente por flechas.

¿Y cómo nos beneficia esto? En pocas palabras, si llamamos a una función con parámetros de menos obtenemos una función **parcialmente aplicada**, es decir una función que toma tantos parámetros como le falte. Utilizar la aplicación parcial de funciones (o llamar a las funciones con menos parámetros) es una forma sencilla de crear funciones al vuelo de forma que podamos pasarlas como parámetros a otras funciones o dotarlas con algunos datos.

Échale un vistazo a esta función ofensivamente simple.

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

¿Qué es lo que realmente pasa cuando realizamos **`multThree 3 5 9`** o **`((multThree 3) 5) 9`**? Primero, 3 es aplicado a **`multThree`** ya que está separado por un espacio. Esto crea una función que toma un parámetro y devuelve una función. Luego 5 es aplicado a está, de forma que se creará una función que toma un parámetro y lo multiplica por 15. 9 es aplicado a esa función y el resultado es 135 o algo similar. Recuerda que el tipo de esta función también podría escribirse como **`multThree :: (Num a) => a -> (a -> (a -> a))`**. Lo que está antes del **`->`** es el parámetro que toma la función y lo que hay después es lo que devuelve. Así que nuestra función toma un **`a`** y devuelve una función con un tipo **`(Num a) => a -> (a -> a)`**. De forma similar, esta función toma una **`a`** y devuelve una función del tipo **`(Num a) => a -> a`**. Y finalmente, esta función toma una **`a`** y devuelve una **`a`**. Mira esto:

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

Al llamar a funciones con menos parámetros de los necesarios, hablando claro, creamos funciones al vuelo ¿Qué pasa si queremos crear una función que tome un número y lo compare con 100? Podríamos hacer algo como esto:

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x
```

Si la llamamos con 99 nos devuelve **`GT`**. Bastante simple. Fíjate en la **`x`** del lado derecho de la ecuación. Ahora vamos a pensar que devuelve`**`compare 100`**. Devuelve una función que toma un número y lo compara con 100. ¡Wau! ¿No es eso lo que buscábamos? Podemos reescribirlo como:

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred = compare 100
```

La declaración de tipo permanece igual ya que **`compare 100`** devuelve una función. **`compare`** tiene el tipo **`(Ord a) => a -> (a -> Ordering)`** y llamarla con 100 devuelve **`(Num a, Ord a) => a -> Ordering`**. La restricción de clase adicional se añade porque 100 es parte también de la clase de tipos **`Num`**.

Nota

¡Asegúrate de que realmente sabes como funcionan las funciones currificadas y la aplicación parcial de funciones ya que son muy importantes!

Las funciones infijas también pueden ser aplicadas parcialmente usando secciones. Para seccionar una función infija simplemente hay que rodearla con paréntesis y suministrar un solo parámetro en un lado. Esto crea una función que toma un parámetro y lo aplica en el lado que falta un operando. Una función extremadamente trivial sería:

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

Llamar a, digamos, **`divideByTen 200`** es equivalente a hacer **`200/10`** o **`(/10) 200`**. Una función que comprueba si un carácter está en mayúsculas sería:

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

Lo único especial de las secciones es el uso de `-`. Por definición, `(-4)` sería una función que toma un número y le restase 4. Sin embargo, por conveniencia, `(-4)` significa menos cuatro. Así que si quieres una función que reste 4 a un número puedes usar `(subtract 4)` o `((-) 4)`.

¿Qué pasa si intentamos hacer `multThree 3 4``en GHCi en lugar de darle un nombre con un ``let` o pasarlo a otra función?

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (t -> t))
      arising from a use of `print' at <interactive>:1:0-12
    Possible fix: add an instance declaration for (Show (t -> t))
    In the expression: print it
    In a 'do' expression: print it
```

GHCi nos está diciendo que expresión producida es una función del tipo `a -> a` pero no sabe como mostrarlo por pantalla. Las funciones no son miembros de la clase de tipos `Show`, así que no podemos obtener una cadena con la representación de una función. Si hacemos algo como `1 + 1` en GHCi, primero calcula que eso es `2`, y luego llama a `show` en `2` para tener una representación textual de ese número. Y una representación textual de `2` es simplemente `"2"`, que es lo que obtenemos por pantalla.

Orden superior en su orden

Las funciones pueden tomar funciones como parámetros y también devolver funciones. Para ilustrar esto vamos a crear una función que tome una función y la aplique dos veces a algo.

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

Primero fíjate en su declaración de tipo. Antes, no necesitábamos usar paréntesis ya que `->` es naturalmente asociativo por la derecha. Sin embargo, aquí está la excepción. Esto indica que el primer parámetro es una función que toma algo y devuelve algo del mismo tipo. El segundo parámetro es algo de ese mismo tipo y también devuelve algo de ese tipo. También podríamos leer esta declaración de tipo de forma currificada, pero para salvarnos de un buen dolor de cabeza diremos simplemente que esta función toma dos parámetros y devuelve una sola cosa. El primer parámetro es una función (del tipo `a -> a`) y el segundo es del mismo tipo `a`. La función puede ser del tipo `Int -> Int` o del tipo `String -> String` o cualquier otra cosa. Pero entonces, el segundo parámetro debe ser del mismo tipo.

Nota

De ahora en adelante diremos que una función toma varios parámetros en lugar de decir que en realidad una función toma un parámetro y devuleve una función parcialmente aplicada hasta que alcance una función que devuleva un valor sólido. Así que para simplificar diremos que `a -> a -> a` toma dos parámetros, incluso aunque nosotros sepamos lo que realmente está pasando.

El cuerpo de la función es muy simple. Usamos el parámetro `f` como una función, aplicando `x` a ella separándolas con un espacio y luego aplicando el resultado a `f` otra vez. De todas formas, juega un poco con la función:

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
```

```
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

Lo increíble y útil de la aplicación parcial es evidente. Si nuestra función requiere que le pasemos una función que tome un solo parámetro, podemos simplemente aplicar parcialmente una función hasta el que tome un solo parámetro y luego pasarla.

Ahora vamos a usar la programación de orden superior para implementar una útil función que está en la librería estándar. Se llama **`zipWith`**. Toma una función y dos listas y las une aplicando la función entre los correspondientes parámetros. Aquí tienes como la implementaríamos:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

Mira la declaración de tipo. El primer elemento es una función que toma dos cosas y produce una tercera. No tienen que ser del mismo tipo, aunque pueden serlo. El segundo y el tercer parámetro son listas. La primera tiene que ser una lista de **`a`** ya que la función de unión toma **`a`** como primer parámetro. La segunda es una lista de **`b`**. El resultado es una lista de **`c`**. Si la declaración de tipo de una función dice que acepta una función **`a -> b -> c`** como parámetro, también aceptará una función del tipo **`a -> a -> a`**. Recuerda que cuando estas creando una función, especialmente de orden superior, y no estas seguro de su tipo, simplemente puedes omitir la declaración de tipo y luego mirar el tipo que infiere Haskell usando **`:t`**.

La acción de la función es muy similar a la de **`zip`**. El caso base es el mismo, solo que hay un parámetro extra, la función de unión, pero este parámetro no tiene importancia en el caso base así que usamos **`_`** con él. El cuerpo de la función para el último patrón es también muy similar al de **`zip`**, solo que no hace **`(x, y)`** sino **`f x y`**. Una sola función de orden superior puede ser utilizada para realizar una multitud de tareas diferentes si es suficientemente general. Aquí tienes una pequeña muestra de las cosas que puede hacer **`zipWith'`**:

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

Como puedes ver, una sola función de orden superior puede ser usada de forma muy versátil. Los lenguajes imperativos usan normalmente cosas como bucles **`while`**, estableciendo alguna variable, comprobando su estado, etc. para conseguir un comportamiento similar y luego envolverlo con una interfaz, una función. La programación funcional utiliza las funciones de orden superior para abstraer los patrones comunes, como examinar dos listas por pares y hacer algo con esos pares o tomar un conjunto de soluciones y eliminar aquellas que no necesites.

Vamos a implementar otra función que ya está en la librería estándar llamada **`flip`**. **`flip`** toma una función y devuelve una función que es como nuestra función original, solo que los dos primeros parámetros están intercambiados. Podemos implementarla así:

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

Aquí, nos aprovechamos del hecho de que las funciones estén currificadas. Cuando llamamos a **`flip'`** sin los parámetros **`x`** e **`y`**, devolverá una función que tome esos parámetros pero los llamará al revés. Incluso aunque las funciones a las que se les ha aplicado **`flip`** son normalmente pasadas a otras funciones, podemos tomar ventaja de la currificación cuando creemos funciones de orden superior pensando de antemano y escribir su resultado final como si fuesen llamadas totalmente aplicadas.

```
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

Asociaciones y filtros

`map` toma una función y una lista y aplica esa función a cada elemento de esa lista, produciendo una nueva lista. Vamos a ver su definición de tipo y como se define.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

La definición de tipo dice que toma una función y que a su vez esta toma un **`a`** y devuelve un **`b`**, una lista de **`a`** y devuelve una lista de **`b`**. Es interesante que simplemente mirando la definición de tipo de una función, a veces podemos decir que hace la función. **`map`** es una de esas funciones de orden superior que son realmente versátiles y que pueden ser usadas de millones formas diferentes. Aquí lo tienes en acción:

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Probablemente te hayas dado cuenta de cada una de estas sentencias se puede conseguir usando listas por comprensión. **`map (+3) [1,5,3,1,6]`** es lo mismo que escribir **`[x+3 | x <- [1,5,3,1,6]]`**. Sin embargo usar **`map`** es mucho más legible cuando solo tienes que aplicar una función a los elementos de una lista, especialmente cuando estas tratando con mapeados de mapeados de modo que se llena todo con un montón de corchetes y termine todo siendo un lío.

`filter` es una función que toma un predicado (un predicado es una función que dice si algo es cierto o falso, o en nuestro caso, una función que devuelve un valor booleano) y una lista y devuelve una lista con los elementos que satisfacen el predicado. La declaración de tipo y la implementación serían algo como:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Bastante simple. Si **`p x`** se evalúa a **`True`** entonces el elemento es incluido en la nueva lista. Si no, se queda fuera. Algunos

ejemplos:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same"
"GAYBALLS"
```

Todo esto podría haberse logrado también con listas por comprensión que usaran predicados. No hay ninguna regla que diga cuando usar **`map`** o **`filter`** en lugar de listas por comprensión, simplemente debes decidir que es más legible dependiendo del contexto. El filtro equivalente de aplicar varios predicados en una lista por comprensión es el mismo que aplicar varios filtrados o unir los predicados usando la función lógica **`&&`**.

¿Recuerdas nuestra función *quicksort* del capítulo anterior? Usamos listas por comprensión para filtrar los elementos que eran menores o iguales y mayores que el pivote. Podemos conseguir lo mismo de forma más legible usando **`filter`**.

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort (filter (<=x) xs)
        biggerSorted = quicksort (filter (>x) xs)
    in  smallerSorted ++ [x] ++ biggerSorted
```

Mapear y filtrar son el pan de cada día de todas las herramientas de un programador funcional. No importa si utilizas las funciones **`map`** y **`filter`** o listas por comprensión. Recuerda como resolvimos el problema de encontrar triángulos rectos con una determinada circunferencia. En programación imperativa, deberíamos haber solucionado el problema anidando tres bucles y luego comprobar si la combinación actual satisface las propiedades de un triángulo recto. En ese caso, lo habríamos mostrado por pantalla o algo parecido. Con la programación funcional este patrón se consigue con el mapeado y filtrado. Creas una función que tome un valor y produzca un resultado. Mapeamos esa función sobre todos los elementos de la lista y luego filtramos la lista resultante para que satisfaga nuestra búsqueda. Gracias a la evaluación perezosa de Haskell, incluso si mapeas algo sobre una lista varias veces o la filtras varias veces, solo se recorrerá la lista una vez.

Vamos a buscar el **número más grande por debajo de 100.000 que sea divisible por 3829**. Para lograrlo, simplemente filtramos un conjunto de posibilidades en el cual sabemos que está la solución.

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

Primero creamos una lista de números menores que 100.000 de forma descendiente. Luego la filtramos con nuestro predicado y como los número están ordenados de forma descendiente, el número más grande que satisface nuestro predicado es el primer elemento de la lista filtrada. Ni siquiera tenemos que usar una lista finita para nuestro conjunto de partida. La evaluación perezosa aparece otra vez. Como al final solo acabamos usando la cabeza de la lista, no importa si la lista es finita o infinita. La evaluación se para cuando se encuentre la primera solución adecuada.

A continuación, vamos a buscar la **suma de todos los cuadrados impares que son menores de 10.000**. Pero primero, como vamos a usarla en nuestra solución, vamos a introducir la función **`takeWhile`**. Toma un predicado y una lista y recorre la lista desde el principio y devuelve estos elementos mientras el predicado se mantenga cierto. Una vez encuentre un predicado que no se evalúe a cierto para. Si queremos obtener la primera palabra de **`"Los elefantes saben como montar una fiesta"`**, podríamos hacer **`takeWhile (/=' ') "Los elefantes saben como montar una fiesta"`** y obtendríamos **`"Los"`**. Vale, ahora a por la suma de todos los cuadrados impares menores que 10.000. Primero empezaremos mapeado la función **`(^2)`** a la lista infinita **`[1..]`**. Luego filtramos la lista para quedarnos solo con los impares. Después tomamos los elementos mientras sean menores que 10.000. Finalmente, obtenemos la suma de todos estos elementos. Ni siquiera tenemos que crear una función para obtener el resultado, podemos hacerlo en una línea en GHCi:

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

¡Impresionante! Empezamos con algunos datos iniciales (la lista infinita de los números naturales) que mapeamos, los filtramos y luego recortamos hasta que encajen con nuestras necesidades para luego sumarlos. También podríamos haber escrito esto usando listas por comprensión.

```
ghci> sum (takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)])
166650
```

Es una cuestión de gustos. De nuevo, la carácterística evaluación perezosa de Haskell es lo que hace esto posible. Podemos mapear y filtrar una lista infinita ya que en realidad ni la mapeará ni la filtrará hasta el final, retrasará dichas acciones. Solo cuando forzamos a Haskell a que nos muestre la suma realiza la suma de que dice a **`takeWhile`** que necesita esos números. **`takeWhile`** fuerza el mapeado y el filtrado, pero solo hasta que encuentre un número mayor o igual que 10.000.

En nuestro siguiente problema vamos tratar con las secuencias de Collatz. Tomamos un número natural. Si ese número es par lo dividimos por dos. Si es impar, lo multiplicamos por tres y le sumamos uno. Tomamos el número resultante y le aplicamos lo mismo, lo que produce un nuevo número y así sucesivamente. Resumiendo, obtenemos una secuencia de números. Se sabe que para todo número la secuencia termina con el uno. Así que empezamos con el número 13, obtenemos esta secuencia: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. 13 * 3 + 1 es igual a 40. 40 dividido por dos es 20, etc. Podemos ver que la secuencia tiene 10 términos. Ahora, lo que queremos saber es: para cada número entre el 1 y el 100 ¿Cuántas secuencias tienen una longitud mayor que 15? Antes de nada creamos una función que produzca una secuencia:

```
chain :: (Integral a) => a -> [a]
chain 1 = [1]
chain n
    | even n =  n:chain (n `div` 2)
    | odd n  =  n:chain (n*3 + 1)
```

Como la secuencia termina en 1, ese es el caso base. Es una función típica recursiva.

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

¡Bien! Parece que funciona correctamente. Y ahora, la función que nos da la respuesta a nuestro problema:

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
```

```
    where isLong xs = length xs > 15
```

Mapeamos con la función **chain** la lista **[1..100]** para obtener la lista de las secuencias. Luego filtramos la lista con un predicado que simplemente nos dice si una lista tiene un tamaño mayor que 15. Una vez hemos realizado el filtrado, vemos cuantas secuencias han quedado en la lista resultante.

Nota

Esta función tiene el tipo **numLongChains :: Int** porque length devuelve el tipo **Int** en lugar de un **Num** por razones históricas.

También podemos hacer cosas como **map (*) [0..]**, con el único motivo de ilustrar como funciona la currificación y como la funciones (parcialmente aplicadas) son valores reales que pueden ser pasadas como parámetros en otras funciones o como pueden ser incluidas en listas (solo que no puedes mostrarlas por pantalla). Hasta ahora solo hemos mapeado sobre listas funciones que toman un solo parámetro, como **map (*2) [0..]** para obtener una lista del tipo **(Num a) => [a]**, pero también podemos usar **map (*) [0..]** sin ningún problema. Lo que sucede es que cada número de la lista es aplicado a ***** que tiene el tipo **(Num a) => a -> a -> a**. Aplicar un solo parámetro a una función que tiene dos parámetros obtenemos una función que solo toma un parámetro, así que tendríamos una lista de funciones **(Num a) => [a -> a]**. **map (*) [0..] `` produce una lista que podríamos escribir como ``[(0*),(1*),(2*),(3*),(4*),(5*)...**

```
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
20
```

Al obtener el 4º elemento de nuestra lista obtenemos una función equivalente a **(4*)**. Y luego aplicamos 5 a esa función. Así que en realidad es como si escribiéramos **(4*) 5** o simplemente **4 * 5**.

Lambdas

Las lambdas son funciones anónimas que suelen ser usadas cuando necesitamos una función una sola vez. Normalmente creamos funciones lambda con el único propósito de pasarlas a funciones de orden superior. Para crear una lambda escribimos un **** (Porque tiene un cierto parecido con la letra griega lambda si le echas mucha imaginación) y luego los parámetros separados por espacios. Luego escribimos una **->** y luego el cuerpo de la función. Normalmente las envolvemos con paréntesis ya que de otra forma se extenderían al resto de la línea.

Si miras 10 cm arriba verás que usamos una sección **where** en nuestra función **numLongChains** para crear la función **isLong** con el único propósito de usarla en un filtro. Bien, en lugar de hacer eso podemos usar una lambda:

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15) (map chain [1..100]))
```

Las lambdas son expresiones, ese es el porqué podemos simplemente pasarlas así. La expresión **(\xs -> length xs > 15)** devuelve una función que nos dice si el tamaño de una lista es mayor que 15.

Es muy común que la gente que no está muy acostumbrada a como funciona la currificación y la aplicación parcial usen lambdas cuando no deben. Por ejemplo, la expresión **map (+3) [1,6,3,2]** y **map (\x -> x + 3) [1,6,3,2]** son equivalentes ya

que ambas expresiones, **(+3)** y **(\x -> x + 3)** son funciones que toman un número y le suman 3. Nada más que decir, crear una lambda en este caso es algo estúpido ya que la aplicación parcial es mucho más legible.

Al igual que las funciones normales, las lambdas pueden tomar cualquier número de parámetros.

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

Y al igual que la funciones normales, las lambdas pueden usar el ajuste de patrones. La única diferencia es que no puedes definir varios patrones para un parámetro, como crear **[]** y **(x:xs)** para el mismo parámetro de forma que las variables se ajusten a uno u a otro. Si el ajuste de patrones falla en una lambda, se lanzará un error de ejecución, así que ten cuidado cuando los uses.

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

Normalmente rodeamos las lambdas con paréntesis a no ser que queramos que se extiendan hasta el final de la línea. Aquí tienes algo interesante, debido a que las funciones se currifican por defecto, estas dos definiciones son iguales:

```
addThree :: (Num a) => a -> a -> a -> a
addThree x y z = x + y + z


addThree :: (Num a) => a -> a -> a -> a
addThree = \x -> \y -> \z -> x + y + z
```

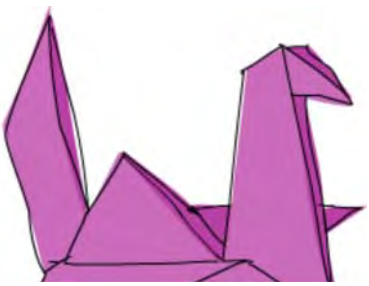
Si definimos funciones de esta forma es obvio el motivo por el cual las definiciones de tipo son como son. Hay tres **->** tanto en la declaración de tipo como en la ecuación. Pero por supuesto, la primera forma de escribir funciones es mucho más legible, y la segundo sirve únicamente para ilustrar la currificación.

Sin embargo hay veces que es más interesante usar esta notación. Creo que la función **flip** es mucho más legible si la definimos así:

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

Aunque es lo mismo que escribir **flip' f x y = f y x**, hacemos obvio que la mayor parte del tipo la usaremos para producir una nueva función. El caso de uso más común de **flip** es llamarla con solo la función parámetro y luego pasar la función resultante como parámetro a **map``o ``filter**. Así que usa las lambdas cuando quieras hacer explícito que tu función esta principalmente pensada para se parcialmente aplicada y se pasada como a una función como parámetro.

Pliegues y papiroflexia



Volviendo a cuando tratábamos con la recursión, nos dimos cuenta de que muchas funciones operaban con listas. Solíamos tener un caso base que era la lista vacía. Debíamos usar un patrón **x:xs** y hacíamos alguna operación con un solo elemento de la lista. Esto sugiere que es un patrón muy común, así que unas cuantas funciones muy útiles fueron creadas para encapsular este comportamiento. Estas funciones son llamadas pliegues (o *folds* en ingles). Son una especie de función **map**, solo que reducen

la lista a un solo valor.

Un pliegue toma una función binaria, un valor inicial (a mi me gusta llamarlo el acumulador) y una lista que plegar. La función binaria toma dos parámetros por si misma. La función binaria es llamada con el acumulador y el primer (o último) elemento y produce un nuevo acumulador. Luego, la función binaria se vuelve a llamar junto al nuevo acumulador y al nuevo primer (o último) elemento de la lista, y así sucesivamente. Cuando se ha recorrido la lista completa, solo permanece un acumulador, que es el valor al que se ha reducido la lista.

Primero vamos a ver la función **`foldl`**, también llamada pliegue por la izquierda. Esta pliega la lista empezando desde la izquierda. La función binaria es aplicada junto a el valor inicial y la cabeza de la lista. Esto produce un nuevo acumulador y la función binaria es vuelta a llamar con ese nuevo valor y el siguiente elemento, etc.

Vamos a volver a implementar **`sum`**, solo que esta vez, vamos a usar un pliegue en lugar de una recursión explícita.

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

Probando, un, dos, tres:

```
ghci> sum' [3,5,2,1]
11
```

Vamos a dar un vistazo a como funciona este pliegue. **`\acc x -> acc + x`** es la función binaria. **`0`** es el valor inicial y **`xs`** es la lista que debe ser plegada. Primero, **`0`** se utiliza como el parámetro **`acc`** en la función binaria y **`3`** es utilizado como el parámetro **`x`** (o el valor actual).``` `0 + 3` ``` produce un **`3`** que pasa a ser el nuevo acumulador. Luego, **`3`** es usado como acumulador y **`5`** como el elemento actual y por tanto **`8`** se convierte en el nuevo acumulador. Seguimos adelante y **`8`** es el acumulador, **`2`** el elemento actual, así que el nuevo acumulador es **`10`**. Para terminar ese **`10`** es usado como acumulador y **`1`** como el elemento actual, produciendo un **`1`**. ¡Enhorabuena, has hecho un pliegue!

A la izquierda tienes un diagrama profesional que ilustra como funciona un pliegue paso a paso. Los números verdes (si los ves amarillos quizás seas daltónico) son los acumuladores. Puedes ver como la lista es consumida por el acumulador de arriba a abajo. Ñam, ñam, ñam... Si tenemos en cuenta que las funciones están currificadas, podemos escribir esta implementación de forma más bonita como:

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

La función lambda **`(\acc x -> acc + x)`** es lo mismo que **`(+)`**. Podemos omitir el parámetro **`xs`** ya que al llamar a **`foldl (+) 0`** nos devuelve una función que toma una lista. Generalmente, si tienes una función del tipo **`foo a = bar b a`** la puedes escribir como **`foo = bar b`** gracias a la currificación.

Vamos a implementar otra función con un pliegue por la izquierda antes de continuar con los pliegues por la derecha. Estoy seguro de que sabes que **`elem`** comprueba si un elemento es parte de una lista así que no lo explicaré de nuevo (mmm... creo que ya lo hice). Vamos a implementarla.

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
```

Bueno, bueno, bueno... ¿Qué estamos haciendo aquí? El valor de inicio y el acumulador son ambos del tipo booleano. Cuando hablamos de pliegues tanto el tipo del acumulador como el tipo del resultado final son el mismo. Empezamos con el valor inicial **`False`**. Tiene sentido ya que asumimos que el elemento no está en la lista. También porque si llamamos a un pliegue con una lista vacía el resultado será simplemente el valor inicial. Luego comprobamos si el elemento actual es el que estamos buscando. Si lo es, ponemos el acumulador a **`True`**. Si no lo es, dejamos el acumulador como estaba. Si ya estaba a **`False`**, permanece en ese estado ya que el elemento actual no es el que buscamos. Si era **`True`**, se queda como estaba también.

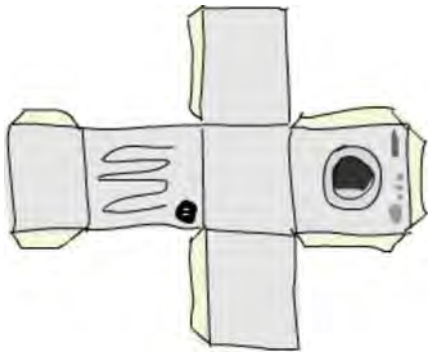
Ahora los pliegues por la derecha funcionan igual que los pliegues por la izquierda, solo que el acumulador consume elemento por la derecha. La función binaria de los pliegues por la izquierda como primer parámetro el acumulador y el valor actual como segundo parámetro (tal que así: **`\acc x -> ...`**), la función binaria de los pliegues por la derecha tiene el valor actual como primer parámetro y el acumulador después (así: **`\x acc -> ...`**). Tiene sentido ya que el pliegue por la derecha tiene el acumulador a la derecha.

El acumulador (y por tanto del resultado) de un pliegue puede ser de cualquier tipo. Puede ser un número, un booleano e incluso una nueva lista. Vamos a implementar la función **`map`** con un pliegue por la derecha. El acumulador será una lista, en la que iremos acumulando los elemento de la lista ya mapeados. Es obvio que el valor inicial será una lista vacía.

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

Si estamos mapeando **`(+3)`** a **`[1,2,3]`**, recorremos la lista desde el lado derecho. Tomamos el último elemento, el cual es **`3`** y le aplicamos la función a él, de forma que acaba siendo un **`6`**. Luego lo añadimos al acumulador que es **`[]`**. **`6:[]`** es **`[6]`** que pasa a ser el nuevo acumulador. Aplicamos **`(+3)`** a **`2`** , que es **`5`** y es añadido (**`:`**) al acumulador, de forma que nos queda **`[5,6]`**. Hacemos lo mismo con el último elemento y acabamos obteniendo **`[4,5,6]`**.

Por supuesto, también podríamos haber implementado esta función usando un pliegue por la izquierda. Sería algo como **`map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs`**, pero la cuestión es que la función **`++`** es bastante menos eficiente que **`:`**, así que normalmente usamos pliegues por la derecha cuando construimos listas a partir de una lista.



Si pones del revés una lista, puedes hacer un pliegue por la derecha como si fuera un pliegue por la izquierda y viceversa. A veces ni siquiera tienes que hacerlo. La función **`sum`** por ejemplo puede ser implementada tanto con un pliegue por la izquierda como por la derecha. Una gran diferencia es que los pliegues por la derecha funcionan con listas infinitas, mientras que los pliegues por la izquierda no. Para aclarar las cosas, si tomas una lista infinita en algún lugar y le aplicas un pliegue por la derecha, en algún momento alcanzará el inicio de la lista. Si embargo, si tomas una lista infinita en algún punto y le aplicas un pliegue por la izquierda nunca alcanzará el final.

Los pliegues se pueden utilizar para implementar cualquier función que recorra una lista, elemento a elemento, y luego devuelvan un valor. Siempre que quieras recorrer una lista y devolver un valor, hay posibilidades de utilizar un pliegue. Esta es la razón por la que los pliegues, junto a los mapeos y los filtros, son unas de las funciones más útiles de la programación funcional.

Las funciones **`foldl1`** y **`foldr1`** son muy parecidas a **`foldl`** y **`foldr`**, solo que en lugar que no necesitas indicar un valor de

inicio. Asumen que el primer (o el último) elemento de la lista es valor de inicio, luego empiezan a plegar la lista por el elemento siguiente. Esto me recuerda que la función **sum** puede ser implementada como: **sum = foldl1 (+)**. Ya que estas funciones dependen de que la listas que van a plegar tengan al menos un elemento, pueden causar errores en tiempo de ejecución si son llamadas con listas vacías. Por otra parte, tanto **foldl** como **foldr** funcionan bien con listas vacías. Cuando hagas un pliegue piensa bien en como actuar ante una lista vacía. Si la función no tiene sentido al ser llamada con listas vacías probablemente puedas utilizar **foldl1``y ``foldr1** para implementarla.

Con el único motivo de mostrarte lo potente que estas funciones son, vamos a implementar un puñado de funciones estándar usando pliegues:

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []

head' :: [a] -> a
head' = foldr1 (\x _ -> x)

last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

head es mejor implementarla con ajuste de patrones, pero de esta forma puedes ver que incluso se puede implementar con pliegues. Nuestra función **reverse'** está bastante clara, creo. Tomamos como valor de inicio la lista vacía y luego recorremos la lista desde la izquierda y simplemente vamos añadiendo elementos a nuestro acumulador. Al final tenemos la lista al revés. **\acc x -> x : acc** se parece a la función **:** solo que los parámetros están al revés. Por esta razón también podíamos haber escrito esto: **foldl (flip (:)) []**.

Existe otra forma de representar los pliegues por la izquierda y por la derecha. Digamos que tenemos un pliegue por la derecha, una función **f** y un valor de inicio **z**. Si hacemos el pliegue sobre la lista **[3,4,5,6]**, básicamente es como si hiciésemos **f 3 (f 4 (f 5 (f 6 z)))**. **f** es llamada con el último elemento de la lista y el acumulador, ese valor es dado como acumulador de la siguiente llamada y así sucesivamente. Si tomamos **+** como **f** y un valor de inicio **0**, tenemos **3 + (4 + (5 + (6 + 0)))**. Representado de forma prefija sería **(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))**. De forma similar si hacemos un pliegue por la izquierda, tomamos **g** como función binaria y **z** como acumulador, sería equivalente a hacer **g (g (g (g z 3) 4) 5) 6**. Si tomamos **flip (:)** como función binaria y **[]** como el acumulador (de forma que estamos poniendo al reverso la lista), entonces sería equivalente a **flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6**. Y estoy casi seguro que si evalúas esta expresión obtendrás **[6,5,4,3]**.

scanl y **scanr** son como **foldl** y **foldr**, solo que devuelven todos los acumuladores intermedios en forma de lista. Existen también **scanl1** y **scanr1**, que son similares a **foldl1** y **foldr1**.

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

Cuando usamos **scanl**, el resultado final será el último elemento de la lista resultante mientras que con **scanr** estará al principio.

Estas funciones son utilizadas para monitorizar la progresión de una función que puede ser implementada con un pliegue. Vamos a contestar a la siguiente cuestión ¿Cuántos elemento toma la suma de todos las raíces de todos los números naturales exceder 1000? Para obtener las raíces de todos los número naturales simplemente hacemos **map sqrt [1..]**. Ahora, para obtener la suma podría utilizar un pliegue, pero como estamos interesados en la progresión de la suma, utilizaremos **scanl**. Cuando obtengamos la lista resultante, simplemente contamos cuantas sumas están por debajo de 1000. La primera suma de la lista será 1. La segunda será 1 más la raíz de 2. La tercera será lo mismo que la anterior más la raíz de 3. Si hay X sumas menores de 1000, entonces tomará X + 1 elementos para que la suma exceda 1000.

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

Utilizamos **takeWhile** en lugar de **filter** porque éste no funciona con listas infinitas. Incluso aunque nosotros sepamos que la lista es ascendente, **filter** no lo sabe, así que usamos **takeWhile** para cortar la lista por la primera ocurrencia de una suma que supere 1000.

Aplicación de funciones con $

Esta bien, ahora vamos a ver la función **$**, también llamada aplicación de función. Antes de nada vamos a ver como está definida:

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

¿Pero qué...? ¿Para qué queremos un operador tan inútil? ¡Es simplemente la aplicación de una función! Bueno, casi, pero no solo eso. Mientras que la aplicación de funciones normal (un espacio entre dos cosas) tiene un alto orden de precedencia, la función **$** tiene el orden de precedencia más bajo. La aplicación de funciones con el espacio es asociativa a izquierdas (así que **f a b c** es lo mismo que **((f a) b) c**), la aplicación de funciones con **$** es asociativa a derechas.

Eso está muy bien, pero ¿De qué nos sirve esto? Básicamente es una función de conveniencia que utilizamos para no tener que escribir muchos paréntesis. Considera la expresión sum **(map sqrt [1..130])**. Gracias a que **$** tiene un bajo orden de precedencia podemos escribir es misma expresión como **sum $ map sqrt [1..130]**, ahorrándonos que nuestros dedos pulsen esas molestas teclas. Cuando se encuentra un **$**, la expresión a la derecha es aplicada como parámetro a la función de la izquierda. ¿Qué pasa con **sqrt 3 + 4 + 9**? Esta expresión suma 4 más 9 más la raíz de 3. Si lo que queremos es la raíz de **3 + 4 + 9** tenemos que escribir **sqrt (3 + 4 + 9)** o si usamos **$** podemos escribirlo como **sqrt $ 3 + 4 + 9** ya que **$** tiene menor orden de precedencia que cualquier otro operador. Por este motivo podemos imaginar a **$** como una especie de paréntesis abierto que de forma automática añade un cierre al final de la expresión.

¿Qué pasaría con **sum (filter (> 10) (map (*2) [2..10]))**? Bueno, como **$** es asociativo por la derecha, **f (g (z**

x)) sería igual que **f $ g $ z x**. Seguimos adelante y **sum (filter (> 10) (map (*2) [2..10]))** puede ser escrito como **sum $ filter (> 10) $ map (*2) [2..10]**.

Pero aparte de eliminar los paréntesis, la existencia del operador **$** también supone que podemos tratar la aplicación de funciones como una función más. De esta forma, podemos, por ejemplo, mapear una lista de funciones:

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

Composición de funciones

En matemáticas la composición de funciones está definida como: $(f \circ g)x = f(g(x))$, que significa que al componer dos funciones se crea una nueva que, cuando se llama con un parámetro, digamos *x*, es equivalente a llamar a *g* con *x* y luego llamar a *f* con el resultado anterior.

En Haskell la composición de funciones es prácticamente lo mismo. Realizamos la composición de funciones con la función **.**, que está definida como:

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

Fíjate en la declaración de tipo. **f** debe tener como parámetro un valor con el mismo tipo que el valor devuelto por **g**. Así que la función resultante toma un parámetro del mismo tipo que toma **g** y devuelve un valor del mismo tipo que devuelve **f**. La expresión **negate . (-3)** devuelve una función que toma un número, lo multiplica por tres y luego lo niega.

Uno de los usos de la composición de funciones es el de crear funciones al vuelo para ser pasadas a otras funciones. Claro, puedes usar lambdas pero muchas veces la composición de funciones es más clara y concisa. Digamos que tenemos una lista de números y queremos convertirlos todos en negativos. Una forma de hacerlo sería obteniendo primero el número absoluto y luego negándolo, algo así:

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

Fíjate que la función lambda se parece a la definición de composición de funciones. Usando la composición de funciones quedaría así:

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

¡Genial! La composición de funciones es asociativa a derechas, así que podemos componer varias funciones al mismo tiempo. La expresión **f (g (z x))** es equivalente a **(f . g . z) x**. Teniendo esto en cuenta, podemos convertir:

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

En esto:

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

¿Y qué pasa con las funciones que toman varios parámetros? Bueno, si queremos usarlas en la composición de funciones, tenemos que aplicarlas parcialmente de forma que cada función tome un solo parámetro. **`sum (replicate 5` `(max 6.7 8.9))`** se puede escribir como **`(sum . replicate 5 . max 6.7) 8.9`** o como **`sum . replicate 5 . max 6.7 $ 8.9`**. Lo que sucede aquí es: se crea una función que toma **`max 6.7`** y aplica **`replicate 5`** a ella. Luego se crea otra función que toma el resultado de lo anterior y realiza una suma. Finalmente, la función anterior es llamada con **`8.9`**. Normalmente se lee como: Aplica **`8.9`** a **`max 6.7`**, luego aplica **`replicate 5`** y luego aplica **`sum`** al resultado anterior. Si quieres reescribir una expresión con un montón de paréntesis usando la composición de funciones, puedes empezar poniendo el último parámetro de la función más externa después de **`$`** y luego empezar a componer todas las demás funciones, escribiéndolas sin el último parámetro y poniendo **`.`** entre ellas. Si tienes **`replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5]` `[4,5,6,7,8])))`** puedes escribirlo también como **`replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]`**. Si una expresión termina con 3 paréntesis, existen posibilidades de escribir la misma expresión usando 3 composiciones de funciones.

Otro uso común de la composición de funciones es la definición de funciones en el llamado estilo libre de puntos. Echa un vistazo a esta función que escribimos anteriormente:

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

Nota

El término *estilo libre de puntos* (*point-free style* o *pointless style* en inglés) se originó en topología, una rama de las matemáticas que trabaja con espacios compuestos de puntos y funciones entre estos espacios. Así que una función en estilo libre de puntos es una función que no menciona explícitamente los puntos (valores) del espacio sobre los que actua. Este término puede confundir a la gente ya que normalmente el estilo libre de puntos implica utilizar el operador de composición de funciones, el cual se representa con un punto en Haskell.

`xs` está expuesta en ambos lados de la ecuación. Podemos eliminar **`xs`** de ambos lados gracias a la currificación, ya que **`foldl (+) 0`** es una función que toma una lista. Escribir la función anterior como **`sum' = foldl (+) 0`** se llama estilo libre de puntos. ¿Cómo escribimos esto en estilo libre de punto?

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

No podemos eliminar simplemente x de ambos lados. La **`x`** en el cuerpo de la función tiene un paréntesis después de ella. **`cos (max 50)`** no tiene mucho sentido. No puedes calcular el coseno de una función. Lo que hacemos es expresar **`fn`** como una composición de funciones.

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

¡Excelente! Muchas veces una composición de funciones es mucho más concisa y legible, ya que te hace pensar en funciones y como se pasan los parámetros entre ellas en lugar de pensar en los datos y como estos son transformados. Puedes utilizar funciones simples con la composición de funciones para crear funciones mucho más complejas. Sin embargo, muchas veces, escribir una función en estilo libre de puntos pude ser menos legible si la función es muy compleja. Es por eso que se desaconseja el uso de la composición de funciones para cadenas de funciones muy largas. El estilo recomendable para estos casos es usar secciones **`let`** para dar nombres a resultados intermedios, dividiendo el problema en sub-problemas y luego

realizar una composición con todo ellos de forma que si alguien lo lee le encuentre el sentido.

En la sección de mapeos y filtros, solventamos el problema de encontrar la suma de todos los cuadrados impares menores que 10.000. Aquí tienes como se vería la solución si la ponemos en una función:

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Siendo fan de la composición de funciones, probablemente podría haberla escrito como:

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

Sin embargo, si hay posibilidades de que alguien más lea este código, podría escribirlo como:

```
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```

No ganaría ninguna competición de código corto, pero le facilitaría la vida a alguien que tuviera que leerlo.

Módulos

Cargando módulos

Un módulo de Haskell es una colección de funciones, tipos y clases de tipos relacionadas entre sí. Un programa Haskell es una colección de módulos donde el módulo principal carga otros módulos y utiliza las funciones definidas en ellos para realizar algo. Tener el código divido en módulos tiene bastantes ventajas. Si un módulo es lo suficientemente genérico, las funciones que son exportadas pueden ser utilizadas en una gran variedad de programas. Si tu código esta separado en módulos que no dependen mucho entre ellos (también decimos que están débilmente acoplados), luego puedes reutilizarlos. Esto hace que la tarea de programar sea más manejable teniendo ya que está todo dividido en varias partes, cada una con su propio propósito.

La librería estándar de Haskell está dividida en varios módulos, cada uno de ellos está formado por funciones y tipos que de alguna forma están relacionados y sirven para propósito común. Existe un módulo para manipular listas, un módulos para la programación concurrente, un módulo para trabajar con números complejos, etc. Todas las funciones, tipos y clases de tipos con las que hemos trabajado hasta ahora son parte del módulo `Prelude`, el cual es importado por defecto. En este capítulo vamos a ver unos cuantos módulos útiles y sus respectivas funciones. Pero primero, vamos a ver como se importan los módulos.

La sintaxis para importar módulos en un script de Haskell es `import <modulename>`. Debe aparecer antes de que definamos cualquier función, así que las importaciones de módulos suelen estar al principio de los ficheros. Un script puede, obviamente, importar varios módulos. Simplemente hay que poner cada`import` en líneas separadas. Vamos a importar el módulo `Data.List`, el cual contiene un puñado de útiles funciones para trabajar con listas, y utilizaremos una función que exporta dicho módulo para crear una función que nos diga cuantos elementos únicos hay en una lista.

```
import Data.List

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

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

También puedes importar módulos y utilizarlos cuando estamos trabajando con GHCi. Si estamos en una sesión de GHCi y queremos utilizar las funciones que exporta `Data.List` hacemos esto:

```
ghci> :m + Data.List
```

Si queremos cargar varios módulos dentro de GHCi no tenemos porque utilizar`:m +` varias veces, simplemente podemos cargar varios módulos de golpe:

```
ghci> :m + Data.List Data.Map Data.Set
```

Sin embargo, si ya has cargado un script que importa algún módulo, no tienes que utilizar `:m +` para poder utilizarlo.

Si únicamente necesitas algunas funciones de un módulo, puedes seleccionarlas para que solamente se importen dichas funciones. Si queremos importan solo las funciones `nub` y `sort` de `Data.List` hacemos lo siguiente:

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

También puede importar todas las funciones de un módulo excepto algunas seleccionadas. Normalmente esto se utiliza cuando tenemos varios módulos que exportan una función con el mismo nombre y nos queremos librar de una de ellas. Digamos que ya tenemos una función llamada `nub` y queremos importar todas las funciones de `Data.List` excepto la función `nub`:

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

Otra forma de tratar con las colisiones de nombres es con las importaciones cualificadas. El módulo `Data.Map`, que ofrece una estructura de datos para buscar valores por clave, exporta un montón de funciones con nombres iguales a las funciones de `Prelude`, como `filter` o `null`. Así que cuando importamos `Data.Map` y llamamos a `filter`, Haskell no sabe a que función llamar. Aquí tienes como lo solucionamos:

```
import qualified Data.Map
```

De esta forma, si queremos referirnos a la función `filter` de `Data.Map`, tenemos que usar `Data.Map.filter`, mientras que si usamos simplemente`filter` nos estamos referimos al filtro normal que todos conocemos. Escribir `Data.Map` delante de todas las funciones es bastante pesado. Por este motivo podemos renombrar una importación cualificada con algo un poco más corto:

```
import qualified Data.Map as M
```

De esta forma, para referirnos a la función `filter` de `Data.Map` solo tenemos que usar `M.filter`.

Puede usar es útilreferencia para ver que módulos están en la librería estándar. Una forma de obtener información acerca de Haskell es simplemente hacer click por la referencia de la librería estándar y explorar sus módulos y sus funciones. También puedes ver el código fuente de cada módulo. Leer el código fuente de algunos módulos es una muy buena forma de aprender Haskell.

Puedes buscar funciones o buscar donde están localizadas usandoHoogle. Es un increíble motor de búsqueda de Haskell. Puedes buscar por nombre de función, nombre de módulo o incluso por la definición de tipo de una función.

Data.List

El módulo `Data.List` trata exclusivamente con listas, obviamente. Ofrece funciones muy útiles para trabajar con listas. Ya hemos utilizado alguna de estas funciones (como `map` y `filter`) ya que el módulo `Prelude`exporta algunas funciones de `Data.List` por conveniencia. No hace falta importar el módulo `Data.List` de forma cualificada porque no colisiona con ningún nombre de `Prelude` excepto por los que ya toma este de`Data.List`. Vamos a dar un vistazo a algunas funciones que aún no hemos conocido.

- `intersperse` toma un elemento y una lista pone ese elemento entre cada par de elementos de la lista. Una demostración:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

- `intercalate` toma una lista y una listas de listas. Inserta la primera lista entre todas las demás lista, dando como resultado una única lista.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

- `transpose` transpone una lista de listas. Si miras la lista de listas como una matriz 2D, las columnas se convierten en filas y viceversa.

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Supongamos que tenemos los polinomios $3x^2 + 5x + 9$, $10x^3 + 9$ y $8x^3 + 5x^2 + x - 1$ y queremos sumarlos. Podemos usar las listas `[0,3,5,9]`, `[10,0,0,9]` y `[8,5,1,-1]` para representarlos en Haskell. Ahora, para sumarlos lo único que tenemos que hacer es:

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```

Cuando transponemos estas tres listas, las potencias cúbicas están en la primera fila, los cuadrados en la segunda fila, etc. Al mapear `sum` sobre esto produce el resultado que buscamos.

- `foldl'` y `foldl1'` son versiones estrictas de sus respectivas versiones perezosas. Cuando usamos pliegues perezosos sobre listas muy grandes solemos obtener errores de desbordamiento de pila. El motivo de que suceda esto es que dada la naturaleza de los pliegues perezosos, el valor acumulador no es actualizado hasta que se realiza el pliegue. Lo que en realidad pasa es que el acumulador hace un especie de promesa de que él calculará el valor cuando se le pide que produzca un resultado (a esto también se le llama thunk). Esto pasa para cada valor intermedio del acumulador y todos esos *thunks* desbordan la pila. Los pliegues estrictos no sufren de este error ya que van calculando de verdad los valores intermedios según recorren la lista en lugar de de llenar la pila con*thunks*. Ya sabes, si alguna vez te encuentras con errores de desbordamiento de pila mientras realizas un pliegue, prueba estas funciones.

- `concat` aplana una lista de listas en una simple lista con los mismos elementos.

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

Básicamente elimina un nivel de anidamiento. Si quieras aplanar completamente `[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]`, que es una lista de listas de listas, tienes que aplanarla dos veces.

- `concatMap` es lo mismo que hacer primero un mapeado con una función a una lista y concatenar todos los resultados.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

- `and` toma una lista de booleanos y devuelve `True` solo si todos los elementos de la lista son `True`.

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

- `or` es como `and` solo que devuelve `True` solo si existe algún elemento `True` en la lista.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

- `any` y `all` toman un predicado y una lista y comprueban si el predicado se satisface para algún o para todos los elementos respectivamente. Normalmente usamos estas funciones en lugar de tener que mapear un lista y luego usar `and` o `or`.

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

- `iterate` toma una función y un valor inicial. Aplica esa función al valor inicial, luego aplica la función al resultado anterior, luego aplica es misma función al resultado anterior otra vez, etc. Devuelve todos los resultados en forma de lista infinita.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

- `splitAt` toma un número y una lista. Luego divide la lista por el índice indicado y devuelve una dupla con ambas listas.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

- `takeWhile` es una función realmente útil. Toma elemento de una lista mientras el predicado se mantenga a cierto, y luego cuando encuentra un elemento que no satisface el predicado, corta la lista.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Digamos que queremos saber la suma de todas las potencias cúbicas que están por debajo de 10.000. No podemos mapear `(^3)` a `[1..]`, aplicar un filtro y luego sumar el resultado ya que filtrar una lista infinita nunca termina. Tu sabes que todos los elementos son ascendentes pero Haskell no lo sabe. Por eso usamos esto:

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

Aplicamos `(^3)` a una lista infinita y una vez que una elemento mayor que 10.000 es encontrado, se corta la lista. De esa forma, luego podemos sumar la lista fácilmente.

- `dropWhile` es similar, solo que descarta elemento mientras el predicado se cumpla. Una vez que el predicado se evalúa a `False`, devuelve el resto de la lista. ¡Una función encantadora!

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

Nos dan una lista que representa los valores de las acciones por fechas. La lista contiene 4-tuplas cuyo primer elemento es el valor de la acción, el segundo el año, el tercero es el mes y el cuarto el día. Si quisiéramos saber cuando una acción alcanzara por primera vez los 1000$, podríamos usar esto:

```
ghci> let stock = [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,20
ghci> head (dropWhile (\(val,y,m,d) -> val < 1000) stock)
(1001.4,2008,9,4)
```

1 - `span` es un tipo de `takeWhile`, solo que devuelve una dupla de listas. La primera lista contiene todo lo que tendría la lista resultante de `takeWhile`. La lista contendría toda la lista que hubiese sido cortada.

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence"
      in "First word:" ++ fw ++ ", the rest:" ++ rest
"First word: This, the rest: is a sentence"
```

- Mientras que `span` divide la lista cuando el predicado deja de cumplirse, `break` divide la lista cuando el predicado se cumple por primera vez. Equivale a `span (not . p)`.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

Cuando usamos `break`, la segunda lista comenzará con el primer elemento que satisfaga el predicado.

- `sort` simplemente ordena una lista. El tipo de elementos que contenga la lista tiene que ser miembro de clase de tipos `Ord`, ya que si los elementos de la lista no se pueden poner en algún tipo de orden, la lista no se puede ordenar.

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

- `group` toma una lista y agrupa los elementos adyacentes que sean iguales en sublistas.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

Si ordenamos una lista antes de agruparla podemos obtener cuantas veces se repite cada elemento.

```
ghci> map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

- `inits` y `tails` son como `init` y `tail`, solo que se aplican recursivamente hasta que no queda nada en la lista. Observa:

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Vamos a usar un pliegue para implementar una búsqueda de una sublista dentro de una lista.

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
    let nlen = length needle
    in  foldl (\acc x -> if take nlen x == needle then True else acc) False (tai
```

Primero llamamos a `tails` con la lista en la que estamos buscando. Luego recorremos cada cola y vemos si empieza con lo que estamos buscando.

- Con esto, en realidad hemos creado una función que se comporta como`isInfixOf`. `isInfixOf` busca una sublista dentro de una lista y devuelve `True` si la sublista que estamos buscando está en algún lugar de la lista.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

- `isPrefixOf` y `isSuffixOf` buscan una sublista desde el principio o des el final de una lista, respectivamente.

2

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
```

```
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

- `elem` y `notElem` comprueban si un elemento está dentro de una lista.

- `partition` toma una lista y un predicado y devuleve una dupla de listas. La primera lista contiene todos los elementos que satisfacen el predicado, la segunda todos los que no.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

  Es importante conocer las diferencias que tiene esta función con `span` y`break`.

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

  Tanto `span` como `break` terminan cuando encuentran el primer elemento que satisface o no satisface el predicado, `partition` recorre la lista entera y la va dividiendo según el predicado.

- `find` toma una lista y un predicado y devuelve el primer elemento que satisface el predicado. Pero, devuelve el elemento envuelto en un valor`Maybe`. Veremos con más detalles los tipos de datos algebraicos en el siguiente capítulo pero de momento esto es todo lo que necesitas saber: un valor `Maybe` puede ser o un `Just algo` o `Nothing`. De la misma forma que una lista puede ser o una lista vacía o una con elementos, un valor `Maybe` puede ser o un elemento o ninguno. Y como el tipo de la lista dice que, por ejemplo, una lista de enteros es `[Int]`, el tipo de un `Maybe` que contenga un entero es `Maybe Int`. De todas formas, vamos a ver la función `find` en acción.

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

  Fíjate en el tipo de `find`. Su resultado es del tipo `Maybe a`. Esto es parecido a tener algo del tipo `[a]`, solo que un valor del tipo`Maybe` solo puede tener o un elemento o ninguno, mientras que una lista puede tener ningún elemento, un solo elemento, o varios de ellos.

  Recuerda cuando estábamos buscando la primera vez que una acción superaba los 1000$. Utilizamos `head (dropWhile (\(val,y,m,d) -> val < 1000)`stock`. Recuerda también que `head` no es una función segura. ¿Qué pasaría si nunca hubiésemos alcanzado los 1000$? `dropWhile` hubiese devuelto una lista vacía y aplicar `head` en una lista vacía solo da un resultado, un error. Sin embargo, si usamos `find (\(val,y,m,d) -> val >1000) stock`, podemos estar mucho más tranquilos. Si nuestras acciones nunca superan los 1000$ (es decir, ningún elemento satisface el predicado), obtendremos `Nothing`, y si sí lo hacen obtendremos una respuesta válida, como `Just (1001.4,2008,9,4)`.

- `elemIndex` es parecido a `elem`, solo que no devuelve un valor booleano. Quizá devuelva el índice del elemento que estamos buscando. Si elemento no está en la lista devolverá `Nothing`.

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
```

```
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

- `elemIndices` es como `elemIndex`, solo que devuelve una lista de índices en caso de que el elemento que estamos buscando aparezca varias veces por la lista. Como estamos usando una lista para representar los índices, no necesitamos el tipo `Maybe`, ya que el caso de que no se encuentre nada puede ser representado con la lista vacía, la cual es sinónimo de `Nothing`.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

- `findIndex` es como `find`, solo que puede devolver el índice del primer elemento que satisfaga el predicado. `findIndices` devuelve el indice de todos los elementos que satisfagan el predicado en forma de lista.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

- Ya hemos hablado de `zip` y de `zipWith`. Vimos que estas funciones combinaban dos listas, ya sea con una dupla o con una función binaria (en el sentido de que toma dos parámetros) ¿Y si queremos combinar tres listas? ¿O combinar tres listas con una función que toma tres parámetros? Bueno, para eso tenemos `zip3`, `zip4`, etc. y `zipWith3`, `zipWith4`, etc. Estas variantes llegan hasta 7. Esto puede parece algún tipo arreglo, pero funciona muy bien en la realidad, ya que no hay tantas ocasiones en las que queramos combinar 8 listas. También existe una forma muy ingeniosa de combinar un número infinito de listas, pero no hemos avanzado aún lo suficiente como para explicarlo aquí.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Como las otras funciones, la listas resultantes son tan largas como la lista más corta.

1

- `lines` es una función muy útil cuando tratamos con algún tipo de entrada, como ficheros. Toma una cadena y devuelve cada línea de esa cadena separada en una lista.

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

`'\n'` es el carácter que representa el salto de línea unix. Las barras invertidas tienen un significado especial en las cadenas y caracteres de Haskell.

- `unlines` es la función inversa de `lines`. Toma una lista de cadenas y las une utilizando un `'\n'`.

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

- `words` y `unwords` sirven para separar separar una línea de texto por palabras. Muy útil.

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these           are    the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

- Ya hemos visto antes `nub`. Toma una lista y elimina los elementos repetidos, devolviendo una lista en la que cada elemento es único. Esta función tiene un nombre muy raro. Resulta que `nub` significa una pequeña parte o una parte esencial de algo. En mi opinión, creo que deberían usar nombres reales para las funciones en lugar de palabras ancestrales.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

- `delete` toma un elemento y una lista y elimina el primer elemento idéntico de esa lista.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

- `\\` es la función división. Funciona como una división basicamente. Elimina la primera ocurrencia de la lista de la derecha de los elementos de la lista de la izquierda.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a  baby"
```

`[1..10] \\ [2,5,9]` es como hacer `delete 2 . delete 5 . delete 9 $[1..10]`.

- `union` funciona como la unión de conjuntos. Devuelve la unión de dos listas. Básicamente recorre cada elemento de la segunda lista y lo añade a la primera lista si está aún no lo contenía. Ten cuidado, los duplicados solo son eliminados de la primera lista.

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

- `intersect` funciona como la intersección de conjuntos. Devuelve los elementos que están en ambas listas.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

- `insert` toma un elemento y una lista que puede ser ordenada e inserta este elemento en la última posición donde sea menor o igual que el elemento siguiente. En otras palabras, `insert` recorrerá la lista hasta que encuentre un elemento mayor que el elemento que estamos insertando, y lo insertará antes de dicho elemento.

```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
```

```
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

El `4` es insertado justo después del `3` y antes del `5` en el primer ejemplo, y entre `3` y el `4` en el segundo.

Si usamos `insert` para introducir algo en una lista ordenada el resultado seguirá estando ordenado.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

Lo que `length`, `take`, `drop`, `splitAt`, `!!` y `replicate` tienen en común es que toman un `Int` como parámetro (o lo devulen), incluso aunque estas funciones podrían ser más genéricas y útiles si simplemente tomaran cualquier tipo que fuera parte de las clases de tipos`Integral` o `Num` (dependiendo de las funciones). Lo hacen por motivos históricos. Probablemente si arreglaran esto dejaría de funcionar mucho código ya existente. Este es el motivo por el que `Data.List` tiene sus propias variantes más genéricas, se llaman `genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex` y `genericReplicate`. Por ejemplo, `length` tiene el tipo `length :: [a] -> Int`. Si intentamos obtener la media de una lista de número usando `let xs = [1..6] in sum xs /length xs` obtendremos un error de tipo, ya que no podemos usar `/` con un`Int`. Por otra parte `genericLength` tiene el tipo `genericLength ::(Num a) => [b] -> a`. Como `Num` puede comportarse como un número en coma flotante, obtener la media haciendo `let xs = [1..6] in sum xs /genericLength xs` funciona perfectamente.

Las funciones `nub`, `delete`, `union`, `intersect` y `group` tienen sus repectivas funciones más generales llamadas `nubBy`, `deleteBy`, `unionBy`, `intersectBy` y `groupBy`. La diferencia entre ellas es que el primer conjunto de funciones usa `==` para comprobar la igualdad, mientras que el otro conjunto toman una función de igualdad y comparan elementos utilizando esta función. `group` es lo mismo que `groupBy (==)`.

Por ejemplo, digamos que tenemos una lista que contiene el valor de una función para cada segundo. Queremos segmentar la lista en sublistas basandonos en cuando un valor estaba por debajo de cero y cuando estaba por encima. Si usaramos un `group` normal simplemente agruparía lod valores iguales adyacentes. Pero lo que nosotros queremos es agruparlos según vaya siendo positivos o no. Aqui es donde entra en juengo `groupBy`. La función de igualdad que toman las funciones con el sufijo `By` deben tomar dos parámetros del mismo tipo y devolver `True` si consideran que son iguales por su propio criterio.

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4, -14.5, 2.9, 2
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

De est forma podemos ver claramente que secciones son positivas y cuales negativas. La función de igualdad que hemos utilizado solo devuleve `True` cuando los dos valores son positivos o son los dos negativos. Esta función de igualdad también se puede escribir como `\x y -> (x > 0) && (y > 0) || (x<= 0) && (y <= 0)` aunque para mi gusto la primera es más legible. Existe incluso una forma más clara de escribir funciones de igualdad para estas funciones si importamos la función `on` del módulo `Data.Function`. `on` se define como:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \x y -> f (g x) (g y)
```

Así que `(==) `on` (> 0)` devuelve una función de igualdad que se comporta igual que `\x y -> (x > 0) == (y > 0)`. `on` se utiliza mucho con todas estas funciones, ya que con ella, podemos hacer cosas como:

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

¡Muy legible! Puedes leerlo de golpe: Agrupa esto por igualdad en si los elementos son mayores que cero.

[1]

De forma similar, la funciones `sort`, `insert`, `maximum` y `minimum` también tienen sus equivalentes más generales. Fucniones como `groupBy` toman funciones que determinan si dos elemento son iguales o no. `sortBy`, `insertBy`, `maximumBy` y `minimumBy` toman una función que determina si si un elemento es mayor, igual o menor que otro. El tipo de `sortBy` es`sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. Si recuerdas, el tipo`Ordering` puede tomar los valores `GT`, `EQ` y `LT`. `sort` es equivalente a `sort compare`, ya que `comapare` simplemente toma dos elementos cuyos tipos esten en la clase de tipos `Ord` y devuelve su relación de orden.

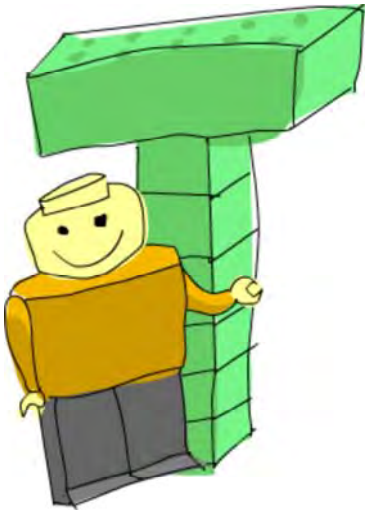
Las listas pueden ser comparadas por orden lexicográfico ¿Y si tenemos una lista de listas y no queremos ordenarlas en base al contenido de las listas interiores sino a sus tamaños? Bueno, como probablemente hayas imaginado, para eso está la función `sortBy`:

```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

¡Increíble! `compare `on` length`, eso se lee casi como el inglés real. Si no estás seguro de como funciona `compare `on` length` aquí, equivalente a`\x y -> length x `compare` length y`. Cuando tratamos con funciones que tienen el sufijo `By` que toman funciones de igualdad normalmente utilizamos`(==) `on` algo` y cuando tratamos con las que toman funciones de orden solemos utilizar `compare `on` algo`.

Data.Char

El módulo `Data.Char` contiene lo que su nombre sugiere. Exporta funciones que tratan con caracteres. También es útil cuando mapeamos o filtramos cadenas ya que al fin y al cabo son listas de caracteres.



`Data.Char` exporta un buen puñado de predicados sobre caracteres. Esto es, funciones que toman un carácter y nos dicen si una suposición acerca de él es verdadera o falsa. Aquí los tienes:

- `isControl` comprueba si un carácter es de control o no.
- `isSpace` comprueba si un carácter es uno de los caracteres de espacio en blanco. Eso incluye espacios, tabuladores, saltos de línea, etc.
- `isLower` comprueba si un carácter está en minúsculas.
- `isUpper` comprueba si un carácter está en mayúsculas.
- `isAlpha` comprueba si un carácter es una letra.
- `isAlphaNum` comprueba si un carácter es una letra o un número.
- `isPrim` comprueba si un carácter es imprimible. Los caracteres de control, por ejemplo, no lo son.
- `isDigit` comprueba si un carácter es un dígito.
- `isOctDigit` comprueba si un carácter es un dígito octal.
- `isHexDigit` comprueba si un carácter es un dígito hexadecimal.
- `isLetter` comprueba si un carácter es una letra.
- `isMark` comprueba si un carácter es una marca Unicode. Esto caracteres que se combinan con sus adyacentes.
- `isNumber` comprueba si un carácter numérico.
- `isPunctuation` comprueba si un carácter es un signo de puntuación.

- `isSymbol` comprueba si un carácter es símbolo matemático o el de una moneda.
- `isSeparator` comprueba si un carácter es un espacio o un separador Unicode.
- `isAscii` comprueba si un carácter es uno de los primeros 128 caracteres del conjunto de caracteres Unicode.
- `isLatin1` comprueba si un carácter es uno de los primeros 256 caracteres del conjunto de caracteres Unicode.
- `isAsciiUpper` comprueba si un carácter está en mayúsculas y además es ascii.
- `isAsciiLower` comprueba si un carácter está en minúsculas y además es ascii.

Todas estas funciones tienen el tipo `Char -> Bool`. La mayoría de las veces las usaras para filtrar cadenas o algo parecido. Por ejemplo, digamos que vamos a hacer un programa que toma un nombre de usuario y dicho nombre solo puede estar compuesto por caracteres alfanuméricos. Podemos usar la función`all` del módulo `Data.List` para determinar si el nombre es correcto:

```
ghci> all isAlphaNum "bobby283"
True
ghci> all isAlphaNum "eddy the fish!"
False
```

En caso de que no te acuerdes, `all` toma un predicado y devuelve `True`solo si dicho predicado se cumple para toda la lista.

También podemos utilizar la función `isSpace` para simular la función`words` del módulo `Data.List`.

```
ghci> words "hey guys its me"
["hey","guys","its","me"]
ghci> groupBy ((==) `on` isSpace) "hey guys its me"
["hey"," ","guys"," ","its"," ","me"]
```

Mmm... bueno, hace lo mismo que `words` pero nos dejamos algunos elementos que contienen un solo espacio ¿Qué podemos hacer? Ya se, vamos a filtrarlos.

```
ghci> filter (not . any isSpace) . groupBy ((==) `on` isSpace) $ "hey guys its me"
["hey","guys","its","me"]
```

`Data.Char` también exporta un tipo de dato parecido a `Ordering`. El tipo`Ordering` puede tener un valor `LT`, `EQ` o `GT`. Es una especie de enumeración. Describe una serie de posibles resultados dados al comparar dos elementos. El tipo `GeneralCategory` también es una enumeración. Representa una serie de categorías a las que puede pertenecer un carácter. La función principal para obtener la categoría de un carácter es `generalCategory`. Tiene el tipo `generalCategory :: Char -> GeneralCategory`. Existen 31 categorías diferentes así que no las vamos a mostrar, pero vamos a jugar un poco con esta función.

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " \t\nA9?|"
[Space,Control,Control,UppercaseLetter,DecimalNumber,OtherPunctuation,MathSymbol]
```

Como `GeneralCategory` forma parte de la clase de tipos `Eq` podemos hacer cosas como `generalCategory c == Space`.

Para terminar, aquí tienes unas cuantas funciones que convierten caracteres:

- `toUpper` convierte un carácter a mayúsculas. Lo espacios, números y todo lo demás permanece igual.

- `toLower` convierte un carácter a minúsculas.

- `toTitle` es similar a `toUpper` excepto para una pocas letras.

- `digitToInt` convierte un carácter a un `Int`. Para que funcione, el carácter debe estar entre los rangos `'0'..'9'`, `'a'..'f'` y `'A'..'F'`.

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
```

- `intToDigit` es la función inversa de `digitToInt`. Toma un `Int` que este en el rango `0..15` y lo convierte a un carácter en minúsculas.

```
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
```

- La función `ord` y `chr` convierte caracteres a sus respectivas representaciones numéricas y viceversa.

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

La diferencia entre dos valores de `ord` de dos caracteres es igual a la diferencia que existe entre ellos dos en la tabla Unicode.

El cifrado César es un método primitivo para cifrar mensajes desplazando cada carácter un número fijo de posiciones en el alfabeto. Podemos crear una especie de cifrado César nosotros mismo, solo que no nos vamos a ceñir únicamente al alfabeto.

```
encode :: Int -> String -> String
encode shift msg =
    let ords = map ord msg
        shifted = map (+ shift) ords
    in  map chr shifted
```

Primero, convertimos la cadena en una lista de número. Luego, le añadimos una cantidad constante a cada número y convertimos la lista de números resultantes en otra cadena de texto. Si te va más la composición, podías haber hecho lo mismo con `map (chr . (+ shift) . ord) msg`. Vamos a probar a codificar unos mensajes.

```
ghci> encode 3 "Heeeeey"
"Khhhhh|"
ghci> encode 4 "Heeeeey"
"Liiiii}"
ghci> encode 1 "abcd"
"bcde"
ghci> encode 5 "Marry Christmas! Ho ho ho!"
"Rfww~%Hmwnxyrfx&%Mt%mt%mt&"
```

Parece que está bien cifrado. Descifrar un mensaje es básicamente volver a poner los caracteres desplazados en su lugar.

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

```
ghci> encode 3 "Im a little teapot"
"Lp#d#olwwoh#whdsrw"
ghci> decode 3 "Lp#d#olwwoh#whdsrw"
"Im a little teapot"
ghci> decode 5 . encode 5 $ "This is a sentence"
"This is a sentence"
```

Data.Map

Las listas de asociación (también llamadas diccionarios) son listas que son utilizadas para almacenar pares clave-valor donde el orden no importa. Por ejemplo, podemos tener una lista de asociación para almacenar números de teléfono, donde los números de telefono serían los valores y los nombres de la gente serían las claves. No nos importa el orden en el que esten almacenados, solo queremos obtener el número aducuado para cada persona.

La forma más obvia de representar una lista de asociación en Haskell sería utilizar una lista de duplas. El primer componente de las duplas sería la clave, y el segundo el valor. Aquí tienes un ejemplo de una lista de asociación de números de telefono:

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

A pesar de que esta alineación extraña, es simplemente un lista de duplas de cadenas. La tarea más común cuando trabajamos con listas de asociación es buscar un valor por clave. Vamos a crear una función que busque un valor dada una clave.

```
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs
```

Muy simple. Esta función toma una clave y una lista, filtra la lista de forma que solo queden claves que coincidan con la clave que se le paso, obtiene el primer elemento de la lista resultante y luego devuelve el valor asociado. Pero ¿Qué pasa si la clave que estamos buscando no esta en la lista? Mmm... Si la clave no está en la lista, acabamos intentando aplicar `head` en una lista vacía, por lo que tendremos un error de ejecución. Sin embargo, debemos evitar que nuestros programas se rompan tan facilmente, así que vamos a usar el tipo `Maybe`. Si no encontramos la clave, devolvemos `Nothing` y en caso de que la encontremos, devolvemos `Just algo`, donde algo es el valor asociado a esa clave.

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
                         then Just v
                         else findKey key xs
```

Fíjate en la declaración de tipo. Toma una clave que puede ser comparada por igualdad, una lista de asociación y puede devolver un valor. Suena bien.

Esta es una función recursiva de libro que opera con listas. Caso base, dividir una lista en cabeza y cola, llamada recursiva... Esto es un pliegue clásico, asi que vamos a implementarlo con un pliegue.

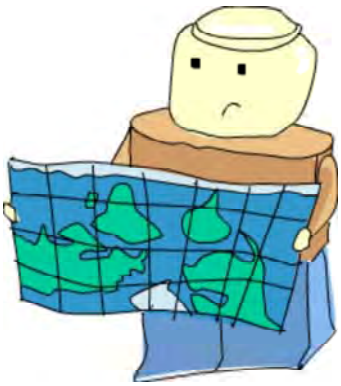
```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

Nota

Normalmente es mejor usar un pligue en estos casos de recursión estándar sobre listas en lugar de una recursión explícita ya que resulta más legible y fácil de identificar. Todo el mundo sabe que se está realizando un pliegue cuando ve una llamada a `foldr`, pero toma un poco más de tiempo leer una recursión explícita.

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```

2



¡Funciona perfectamente! Si tenemos el número de una chica obtenemos dicho `Just número`, en otro caso obtenemos `Nothing`.

Acabamos de implementar la función `lookup` del módulo `Data.List`. Si queremos obtener el valor correspondiente a una clave solo tenemos que recorrer la lista hasta que la encontremos. El módulo `Data.Tree` ofrece listas de asociación mucho más eficientes (ya que están implementadas con árboles) y también ofrece un montón de funciones útiles. De ahora en adelante diremos que estamos trabajando con diccionarios en lugar de listas de asociación.

Debido a que `Data.Map` exporta funciones que colisionan con las de `Prelude` y `Data.List`, lo importaremos de forma cualificada.

```
import qualified Data.Map as Map
```

Pon esta sentencia en un script y luego cárgalo con GHCi.

Vamos a continuar y ver que tiene `Data.Map` para nosotros. Aquí tienes un resumen básico de las funciones.

- La función `fromList` tomo una lista de asociación (en forma de lista) y devuelve un diccionario con las mismas asociaciones.

```
ghci> Map.fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2
fromList [("betty","555-2938"),("bonnie","452-2928"),("lucille","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

En caso de que existan claves duplicadas en la lista de asociación, los duplicados son descartados. Este es la declaración de tipo de `fromList`:

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

Dice que toma una lista de duplas `k` y `v` y devuelve un diccionario que asocia las claves `k` con los valores `v`. Fíjate que cuando creábamos listas de asociación con listas normales, las claves solo tenían que ser igualables (su tipo pertenecía a la clase de tipos `Eq`) pero ahora tienen que ser ordenables. Esto es básicamente una restricción del módulo `Data.Map`. Necesita que las claves sean ordenables para que pueda organizarlas en un árbol.

Debes utilizar siempre el módulo `Data.Map` para las asociaciones clave-valor a no ser que las claves son sean de la clase de tipos `Ord`.

- `empty` representa un diccionario vacío. No toma ningún parámetro, simplemente devuelve un diccionario vacío.

```
ghci> Map.empty
fromList []
```

- `insert` toma una una clave, un valor y un diccionario y devuelve un nuevo diccionario exactamente igual al anterior, solo que contiene además la nueva clave-valor.

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600 (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 . Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

Podemos implementar nuestra propia función `fromList` usando únicamente un diccionario vacío, `insert` y un pliegue. Mira:

```
fromList' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty
```

Es un pliegue bastante simple. Empezamos con un diccionario vacío y luego vamos plegando la lista desde la derecha, insertando nuevos pares clave-valor en el acumulador.

- `null` comprueba si un diccionario está vacío.

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
```

- `size` nos da el tamaño de un diccionario.

```
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]
5
```

- `singleton` toma una clave y un valor y nos devuelve un diccionario que solo contiene esa clave.

```
ghci> Map.singleton 3 9
fromList [(3,9)]
ghci> Map.insert 5 9 $ Map.singleton 3 9
fromList [(3,9),(5,9)]
```

- `lookup` funciona como la función `lookup` de `Data.List`, solo que opera con diccionarios en lugar de listas. Devuelve `Just algo` si encuentra la clave o `Nothing` en caso contrario.

- `member` es un predicado que toma una clave y un diccionario y nos dice si dicha clave está contenida en el diccionario.

```
ghci> Map.member 3 $ Map.fromList [(3,6),(4,3),(6,9)]
True
ghci> Map.member 3 $ Map.fromList [(2,5),(4,5)]
False
```

- `map` y `filter` funcionan de forma similar a sus equivales de listas.

```
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $ Map.fromList [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

- `toList` es la inversa de `fromList`.

  ghci> Map.toList . Map.insert 9 2 $ Map.singleton 4 3 [(4,3),(9,2)]

- `keys` y `elems` devuelven una lista con todas la claves o todo los valores respectivamente. `keys` es equivalente a `map fst . Map.toList`y `elems` es equivalente a `map snd . Map.toList`.

- `fromListWith` es una función muy interesante. Actúa com `fromList`, solo que no descarta ningún predicado, en su lugar, utiliza una función que le pasemos para decidir cual de ellas debe añadirse. Digamos que una chica puede tener varios números de teléfono y tenemos una lista de asociación como esta:

```
phoneBook =
    [("betty","555-2938")
    ,("betty","342-2492")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("patsy","943-2929")
    ,("patsy","827-9162")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ,("penny","555-2111")
    ]
```

De esta forma si usamos `fromList` perderemos algunos números. Así que podemos hacer esto:

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ numl
```

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

En caso de que se encuentre una clave duplicada, la función que pasemos se encargará de combinar los valores de es clave. También podríamos hacer primero todos los valores de la lista de asociación listas unitarias y luego utilizar ++ para combinar los números.

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs


ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162","943-2929","493-2928"]
```

Muy simple. Otro caso sería si estamos creando un diccionario a partir de una lista de asociación que contiene números y que cuando se encuentra una clave duplicada, queremos que el valor más grande sea el que se mantenga.

```
ghci> Map.fromListWith max [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,1
fromList [(2,100),(3,29),(4,22)]
```

O también podríamos haber elegido que estos valores se sumaran:

```
ghci> Map.fromListWith (+) [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,1
fromList [(2,108),(3,62),(4,37)]
```

- `insertWith` es un `insert` de la misma forma que `fromListWith` lo es para `fromList`. Inserta una clave-valor en un diccionario, pero si el diccionario ya contiene dicha clave, usa la función que le pasamos para determinar que hacer.

```
ghci> Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
```

Estas son solo algunas de la funciones que contiene `Data.Map`. Puedes ver un lista completa de las funciones que contiene en sudocumentación.

Data.Set

El módulo `Data.Set` nos ofrece operaciones con conjuntos. Conjuntos como los conjuntos en matemáticas. Los `conjuntos` son un tipo de datos mezcla entre las lista y los diccionarios. Todos los elementos de un conjunto son únicos. Y como internamente son implementados con árboles (como los diccionarios de `Data.Map`) están ordenados. Comprobar si existe un elemento, insertarlo, eliminarlo, etc. es mucho más eficiente que hacerlo con listas. Las operaciones más comunes cuando trabajamos con conjuntos son insertar elementos, comprobar si existe un elemento en el conjunto y convertir un conjunto en una lista.

Como los nombres que exporta `Data.Set` colisionan con los de `Prelude` y`Data.List` lo importamos de forma cualificada.

Pon esta sentencia en un script:

```
import qualified Data.Set as Set
```

Y luego carga el script con GHCi.

Digamos que tenemos dos trozos de texto. Queremos saber que caracteres son usados en ambos trozos.

```
text1 = "I just had an anime dream. Anime... Reality... Are they so different?"
text2 = "The old man left his garbage can out and now his trash is all over my lawn!"
```

La función `fromList` funciona como es de esperar. Toma una lista y la convierte en un conjunto.

```
ghci> let set1 = Set.fromList text1
ghci> let set2 = Set.fromList text2
ghci> set1
fromList " .?AIRadefhijlmnorstuy"
ghci> set2
fromList " !Tabcdefghilmnorstuvwy"
```

Como puedes ver los elementos están ordenados y cada elemento es único. Ahora vamos a utilizar la función `intersection` para ver que elementos están en ambos conjuntos.

```
ghci> Set.intersection set1 set2
fromList " adefhilmnorstuy"
```

Podemos usar la función `difference` para ver que elementos del primer conjunto no están en el segundo y viceversa.

```
ghci> Set.difference set1 set2
fromList ".?AIRj"
ghci> Set.difference set2 set1
fromList "!Tbcgvw"
```

O podemos ver todas letras que fueron utilizadas en ambos textos usando`union`.

```
ghci> Set.union set1 set2
fromList " !.?AIRTabcdefghijlmnorstuvwy"
```

Las funciones `null`, `size`, `member`, `empty`, `singleton`, `insert` y `delete` funcionan como esperas.

```
ghci> Set.null Set.empty
True
ghci> Set.null $ Set.fromList [3,4,5,5,4,3]
False
ghci> Set.size $ Set.fromList [3,4,5,3,4,5]
3
ghci> Set.singleton 9
fromList [9]
ghci> Set.insert 4 $ Set.fromList [9,3,8,1]
fromList [1,3,4,8,9]
ghci> Set.insert 8 $ Set.fromList [5..10]
fromList [5,6,7,8,9,10]
ghci> Set.delete 4 $ Set.fromList [3,4,5,4,3,4,5]
fromList [3,5]
```

También se puede consultar por subconjuntos o conjuntos propios. El conjunto A es un subconjunto de B, si B contiene todos los elementos de A. El conjunto A es un conjunto propio de B si B contiene todos los elementos que contiene A y ninguno más.

```
ghci> Set.fromList [2,3,4] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
True
ghci> Set.fromList [1,2,3,4,5] `Set.isProperSubsetOf` Set.fromList [1,2,3,4,5]
False
```

```
ghci> Set.fromList [2,3,4,8] `Set.isSubsetOf` Set.fromList [1,2,3,4,5]
False
```

También podemos usar las funciones `map` y `filter` con ellos.

```
ghci> Set.filter odd $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,5,7]
ghci> Set.map (+1) $ Set.fromList [3,4,5,6,7,2,3,4]
fromList [3,4,5,6,7,8]
```

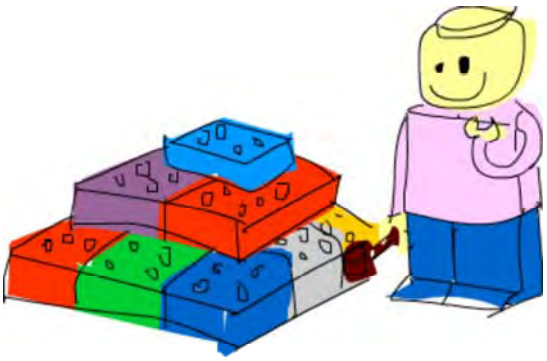
Los conjuntos son normalmente utilizados para eliminar los elementos duplicados de una lista de forma que primero hacemos un conjunto con``fromList` y luego lo volvemos a convertir en una lista con `toList`. La función `nub` de `Data.List` ya realiza esta tarea, pero si estas eliminando duplicados de un gran lista es mucho más eficiente si insertar los elementos en un conjunto y luego convertirlo en una lista en lugar de utilizar`nub`. Pero `nub` solo requiere que los elemento de la lista sean de la clase de tipos `Eq`, mientras que los elementos de los conjuntos deben ser de la clase `Ord`.

```
ghci> let setNub xs = Set.toList $ Set.fromList xs
ghci> setNub "HEY WHATS CRACKALACKIN"
" ACEHIKLNRSTWY"
ghci> nub "HEY WHATS CRACKALACKIN"
"HEY WATSCRKLIN"
```

`setNub` es mucho más rápido que `nub` para listas grandes, pero como puedes ver, `nub` preserva el orden en los que los elementos aparecen en la lista mientras que `setNub` no.

Creando nuestros propios módulos

Hasta ahora hemos visto unos cuantos módulos interesantes, pero ¿Cómo creamos nuestros propios módulos? Casi todo lenguaje de programación te permite que dividas tu código en varios ficheros y Haskell no es diferente. Cuando creamos programas, es una buena práctica que las funciones y tipos que de alguna forma están relacionados estén en el mismo módulo. De esta forma, podemos fácilmente reutilizar esas funciones en otros programas importando esos módulos.



Vamos a ver como podemos crear nuestro propio módulo haciendo un pequeño módulo que exporte funciones que nos permitan calcular el volumen y el área de unos cuantos objetos geométricos. Empezaremos creando un fichero llamado`Geometry.hs`.

Decimos que un módulo exporta unas funciones. Lo que significa que cuando utilizamos un módulo, podemos ver las funciones que dicho modulo exporta. Puede definir funciones que son llamadas internamente, pero solo podemos ver las funciones que exporta.

Especificamos el nombre de un módulo al principio del módulo. Si hemos llamado al fichero `Geometry.hs` debemos darle el nombre de `Geomtry` a nuestro módulo. Luego, especificamos la funciones que se exportan, y luego comenzamos a definir dichas funciones. Así que empezamos con esto.

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
```

```
, cuboidVolume
) where
```

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

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

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

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

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

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

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

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

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

Geometría clásica. Aunque hay un par de cosas que destacar. Como un cubo es un caso especial de un hexaedro, hemos definido el área y el volumen tratándolo como un hexaedro con todos sus lados iguales. También hemos definido una función auxiliar llamada `rectangleArea`, la cual calcula el área de un rectángulo basándose en el tamaño de sus lados. Es muy trivial ya que se trata de una multiplicación. Hemos utilizado esta función en las funciones`cuboidArea` y `cuboidVolume` pero no la hemos exportado. Esto es debido a que queremos que nuestro módulo solo muestre funciones para tratar estos tres objetos dimensionales, hemos utilizado `rectangleArea` pero no la hemos exportado.

Cuando estamos creando un módulo, normalmente exportamos solo las funciones que actúan como una especia de interfaz de nuestro módulo de forma que la implementación se mantenga oculta. Si alguien usa nuestro módulo `Geometry`, no nos tenemos que preocupar por funciones las funciones que no exportamos. Podemos decidir cambiar esas funciones por completo o eliminarlas a cambio de una nueva versión (podríamos eliminar `rectangleArea` y utilizar `*`) y nadie se daría cuenta ya que no las estamos exportando.

Para utilizar nuestro módulos simplemente usamos:

```
import Geometry
```

Aunque `Geometry.hs` debe estar en el mismo directorio que el programa que lo está utilizando.

También podemos dar a los módulos una estructura jerárquica. Cada módulo puede tener cualquier número de submódulos y ellos mismo pueden tener cualquier otro número de submódulos. Vamos a dividir las funciones del módulo `Geometry` en tres submódulos de forma de cada objeto tenga su propio módulo.

Primero creamos un directorio llamado `Geometry`. Mantén la G en mayúsculas. Dentro de él crearemos los ficheros `sphere.hs`, `cuboid.hs`, y`cube.hs`. Este será el contenido de los ficheros:

`sphere.hs`

```
module Geometry.Sphere
( volume
, area
) where

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

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

`cuboid.hs`

```
module Geometry.Cuboid
( volume
, area
) where

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

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

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

`cube.hs`

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

¡Bien! El primero es `Geometry.Sphere`. Fíjate que primero hemos creado una carpeta llamada `Geometry` y luego y luego hemos definido el nombre como`Geometry.Sphere`. Hicimos los mismo con el hexaedro. Fíjate también que en los tres módulos hemos definido funciones con los mismos nombres. Podemos hacer esto porque están separados en módulos distintos. Queremos utilizar las funciones de `Geometry.Cuboid` en `Geometry.Cube` pero no podemos usar simplemente `import Geometry.Cuboid` ya que importaríamos funciones con el mismo nombre que en `Geometry.Cube`. Por este motivo lo cualificamos.

Así que si ahora estamos en un fichero que se encuentra en el mismo lugar que la carpeta `Geometry` podemos utilizar:

```
import Geometry.Sphere
```

Y luego podemos utilizar `area` y `volume` y nos darán el área y el volumen de una esfera. Si queremos usar dos o más módulos de éstos, tenemos que cualificarlos para que no hayan conflictos con los nombres. Podemos usar algo como:

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

Ahora podemos llamar a `Sphere.area`, `Sphere.volume`, `Cuboid.area`, etc. y cada uno calculará el área o el volumen de su respectivo objeto.

La próxima que te encuentres escribiendo un módulo que es muy grande y tienen un montón de funciones, trata de encontrar que funciones tienen un propósito común y luego intenta ponerlas en un mismo módulo. De esta forma, serás capaz de importar dicho módulo la próxima vez que escribas un programa y requiera la misma funcionalidad.

Creando nuestros propios tipos y clases de tipos

En capítulos anteriores vimos algunos tipos y clases de tipos de Haskell. ¡En este capítulo vamos a ver como crearlos nosotros mismos! ¿A qué no te lo esperabas?

Introducción a los tipos de datos algebraicos

Hasta ahora hemos jugado con muchos tipos: `Bool`, `Int`, `Char`, `Maybe`, etc. Pero ¿Cómo los creamos? Bueno, una forma es usar la palabra clave `data` para definir un tipo. Vamos a ver como está definido el tipo`Bool` en la librería estándar:

```
data Bool = False | True
```

`data` significa que vamos a definir un nuevo tipo de dato. La parte a la izquierda del `=` denota el tipo, que es `Bool`. La parte a la derecha son los **constructores de datos**. Estos especifican los diferentes valores que puede tener un tipo. El `|` se puede leer como una *o*. Así que lo podemos leer como: El tipo `Bool` puede tener un valor `True` o `False`. Tanto el nombre del tipo como el de los constructores de datos deben tener la primera letra en mayúsculas.

De la misma forma podemos pensar que el tipo `Int` está definido como:

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647
```

El primer y el último constructor de datos son el mínimo y el máximo valor posible del tipo `Int`. En realidad no está definido así, los tres puntos están ahí porque hemos omitido una buena cantidad de números, así que esto es solo para motivos ilustrativos.

Ahora vamos a pensar en como definiríamos una figura en Haskell. Una forma sería usar tuplas. Un círculo podría ser `(43.1, 55.0, 10.4)` donde el primer y el segundo campo son las coordenadas del centro del círculo mientras que el tercer campo sería el radio. Suena bien, pero esto nos permitiría también definir un vector 3D o cualquier otra cosa. Una solución mejor sería crear nuestro propio tipo que represente una figura. Digamos que una figura solo puede ser un círculo o un rectángulo:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

¿Qué es esto? Piensa un poco a que se parece. El constructor de datos`Circle` tiene tres campos que toman valores en coma flotante. Cuando creamos un constructor de datos, opcionalmente podemos añadir tipos después de él de forma que estos serán los valores que contenga. Aquí, los primeros dos componentes son las coordenadas del centro, mientras que el tercero es el radio. El constructor de datos `Rectangle` tiene cuatro campos que aceptan valores en coma flotante. Los dos primeros representan las coordenadas de la esquina superior izquierda y los otros dos las coordenadas de la inferior derecha.

Ahora, cuando hablamos de campos, en realidad estamos hablando de parámetros. Los constructores de datos son en realidad funciones que devuelven un valor del tipo para el que fueron definidos. Vamos a ver la declaración de tipo de estos dos constructores de datos.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Bien, los constructores de datos son funciones como todo lo demás ¿Quíen lo hubiera pensado? Vamos a hacer una función que tome una figura y devuleva su superficie o área:

```
surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

La primera cosa destacable aquí es la declaración de tipo. Dice que toma una figura y devuelve un valor en coma flotante. No podemos escribir una declaración de tipo como `Circle -> Float` ya que `Circle` no es un tipo,`Shape` si lo es. Del mismo modo no podemos declarar una función cuya declaración de tipo sea `True -> Int`. La siguiente cosa que podemos destacar es que podemos usar el ajuste de patrones con los constructores. Ya hemos utilizado el ajuste de patrones con constructores anteriormente (en realidad todo el tiempo) cuando ajustamos valores como `[]`, `False`, `5`, solo que esos valores no tienen campos. Simplemente escribimos el constructor y luego ligamos sus campos a nombres. Como estamos interesados en el radio, realmente no nos importan los dos primeros valores que nos dicen donde está el círculo.

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Bien ¡Funciona! Pero si intentamos mostrar por pantalla `Circle 10 20 5`en una sesión de GHCi obtendremos un error. Esto sucede porque Haskell aún no sabe como representar nuestro tipo con una cadena. Recuerda que cuando intentamos mostrar un valor por pantalla, primero Haskell ejecuta la función`show` para obtener la representación en texto de un dato y luego lo muestra en la terminal. Para hacer que nuestro tipo `Shape` forme parte de la clase de tipo `Show` hacemos esto:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving (Show)
```

No vamos a preocuparnos ahora mismo acerca de derivar. Simplemente diremos que si añadimos `deriving (Show)` al final de una declaración de tipo, automáticamente Haskell hace que ese tipo forme parte de la clase de tipos`Show`. Así que ahora ya podemos hacer esto:

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Los constructores de datos son funciones, así que podemos mapearlos, aplicarlos parcialmente o cualquier otra cosa. Si queremos una lista de círculos concéntricos con diferente radio podemos escribir esto:

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,Circle 10.0 20.0 6.0]
```

Nuestro tipo de dato es bueno, pero podría se mejor. Vamos a crear un tipo de dato intermedio que defina un punto en espacio bidimensional. Luego lo usaremos para hacer nuestro tipo más evidente.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

Te habrás dado cuenta de que hemos usado el mismo nombre para el tipo que para el constructor de datos. No tiene nada de especial, es algo común usar el mismo nombre que el del tipo si solo hay un constructor de datos. Así que ahora `Circle` tiene dos campos, uno es el del tipo `Point` y el otro del tipo `Float`. De esta forma es más fácil entender que es cada cosa. Lo mismo sucede para el rectángulo. Tenemos que modificar nuestra función `surface` para que refleje estos cambios.

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Lo único que hemos cambiado han sido los patrones. Hemos descartado completamente el punto en el patrón del círculo. Por otra parte, en el patrón del rectángulo, simplemente hemos usado un ajuste de patrones anidado para obtener las coordenadas de los puntos. Si hubiésemos querido hacer una referencia directamente a los puntos por cualquier motivo podríamos haber utilizado un patrón *como*.

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

¿Cómo sería una función que desplaza una figura? Tomaría una figura, la cantidad que se debe desplazar en el eje *x*, la cantidad que se debe desplazar en el eje *y* y devolvería una nueva figura con las mismas dimensiones pero desplazada.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b = Rectangle (Point (x1+a) (y1+b)) (Poin
```

Bastante sencillo. Añadimos las cantidades a desplazar a los puntos que representan la posición de las figuras.

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

Si no queremos trabajar directamente con puntos, podemos crear funciones auxiliares que creen figuras de algún tamaño en el centro del eje de coordenadas de modo que luego las podamos desplazar.

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Como es lógico, podemos exportar nuestros datos en los módulos. Para hacerlo, solo tenemos que escribir el nombre del tipo juntos a las funciones exportadas, y luego añadirles unos paréntesis que contengan los constructores de datos que queramos que se exporten, separados por comas. Si queremos que se exporten todos los constructores de datos para un cierto tipo podemos usar...

Si quisiéramos exportar las funciones y tipos que acabamos de crear en un módulo, podríamos empezar con esto:

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

Haciendo `Shape (..)` estamos exportando todos los constructores de datos de `Shape`, lo que significa que cualquiera que importe nuestro módulo puede crear figuras usando los constructores `Circle` y `Rectangle`. Sería lo mismo que escribir `Shape (Rectangle, Circle)`.

También podríamos optar por no exportar ningún constructor de datos para`Shape` simplemente escribiendo `Shape` en dicha sentencia. De esta forma, quien importe nuestro módulo solo podrá crear figuras utilizando las funciones auxiliares `baseCircle` y `baseRect`. `Data.Map` utiliza este método. No puedes crear un diccionario utilizando `Map.Map [(1,2),(3,4)]` ya que no se exporta el constructor de datos. Sin embargo, podemos crear un diccionario utilizando funciones auxiliares como `Map.fromList`. Recuerda, los constructores de datos son simples funciones que toman los campos del tipo como parámetros y devuelven un valor de un cierto tipo (como `Shape`) como resultado. Así que cuando elegimos no exportarlos, estamos previniendo que la gente que importa nuestro módulo pueda utilizar esas funciones, pero si alguna otra función devuelve devuelve el tipo que estamos exportando, las podemos utilizar para crear nuestros propios valores de ese tipo.

No exportar los constructores de datos de un tipo de dato lo hace más abstracto en el sentido de que oculta su implementación. Sin embargo, los usuarios del módulo no podrán usar el ajuste de patrones sobre ese tipo.

Sintaxis de registro

Bien, se nos ha dado la tarea de crear un tipo que describa a una persona. La información que queremos almacenar de cada persona es: nombre, apellidos, edad, altura, número de teléfono y el sabor de su helado favorito. No se nada acerca de ti, pero para mi es todo lo que necesito saber de una persona. ¡Vamos allá!

```
data Person = Person String String Int Float String String deriving (Show)
```

Vale. El primer campo es el nombre, el segundo el apellido, el tercero su edad y seguimos contando. Vamos a crear una persona.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Parece interesante, pero desde luego no muy legible ¿Y si queremos crear una función que obtenga información por separado de una persona? Una función que obtenga el nombre de una persona, otra función que obtenga el apellido, etc. Bueno, las tendríamos que definir así:

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
```

```
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

¡Fiuuu! La verdad es que no me divertido escribiendo esto. A parte de que este método sea un lío y un poco ABURRIDO de escribir, funciona.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

Ahora es cuando piensas: debe de haber un método mejor. Pues no, lo siento mucho.

Estaba de broma :P Si que lo hay. Los creadores de Haskell fueron muy inteligentes y anticiparon este escenario. Incluyeron un método alternativo de definir tipos de dato. Así es como podríamos conseguir la misma funcionalidad con la sintaxis de registro.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

En lugar de nombrar los campos uno tras otro separados por espacios, utilizamos un par de llaves. Dentro, primero escribimos el nombre de un campo, por ejemplo `firstName` y luego escribimos unos dobles puntos `::`(también conocido como *Paamayim Nekudotayim* xD) y luego especificamos el tipo. El tipo de dato resultante es exactamente el mismo. La principal diferencia es que de esta forma se crean funciones que obtienen esos campos del tipo de dato. Al usar la sintaxis de registro con este tipo de dato, Haskell automáticamente crea estas funciones: `firstName`, `lastName`,`age`, `height`, `phoneNumber` y `flavor`.

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Hay otro beneficio cuando utilizamos la sintaxis de registro. Cuando derivamos`Show` para un tipo, mostrará los datos de forma diferente si utilizamos la sintaxis de registro para definir e instanciar el tipo. Supongamos que tenemos un tipo que representa un coche. Queremos mantener un registro de la compañía que lo hizo, el nombre del modelo y su años de producción. Mira.

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

Si lo definimos usando la sintaxis de registro, podemos crear un coche nuevo de esta forma:

```
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

Cuando creamos un coche nuevo, no hace falta poner los campos en el orden adecuado mientras que los pongamos todos. Pero si no usamos la sintaxis de registro debemos especificarlos en su orden correcto.

Utiliza la sintaxis de registro cuando un constructor tenga varios campos y no sea obvio que campo es cada uno. Si definimos el tipo de un vector 3D como`data Vector = Vector Int Int Int`, es bastante obvio que esos campos son las componentes del vector. Sin embargo, en nuestros tipo `Person` y`Car`, no es tan obvio y nos beneficia mucho el uso de esta sintaxis.

Parámetros de tipo

Un constructor de datos puede tomar algunos valores como parámetros y producir un nuevo valor. Por ejemplo, el constructor `Car` toma tres valores y produce un valor del tipo coche. De forma similar, un **constructor de tipos**puede tomar tipos como parámetros y producir nuevos tipos. Esto puede parecer un poco recursivo al principio, pero no es nada complicado. Si has utilizado las plantillas de *C++* te será familiar. Para obtener una imagen clara de como los parámetros de tipo funcionan en realidad, vamos a ver un ejemplo de como un tipo que ya conocemos es implementado.

```
data Maybe a = Nothing | Just a
```

La `a` es un parámetro de tipo. Debido a que hay un parámetro de tipo involucrado en esta definición, llamamos a `Maybe` un constructor de tipos. Dependiendo de lo que queramos que este tipo contenga cuando un valor no es`Nothing`, este tipo puede acabar produciendo tipos como `Maybe Int`,`Maybe Car`, `Maybe String`, etc. Ningún valor puede tener un tipo que sea simplemente `Maybe`, ya que eso no es un tipo por si mismo, es un constructor de tipos. Para que sea un tipo real que algún valor pueda tener, tiene que tener todos los parámetros de tipo definidos.

Si pasamos `Char` como parámetro de tipo a `Maybe`, obtendremos el tipo`Maybe Char`. Por ejemplo, el valor `Just 'a'` tiene el tipo `MaybeChar`.

Puede que no lo sepas, pero utilizamos un tipo que tenía un parámetro de tipo antes de que empezáramos a utilizar el tipo `Maybe`. Ese tipo es el tipo lista. Aunque hay un poco decoración sintáctica, el tipo lista toma un parámetro para producir un tipo concreto. Los valores pueden tener un tipo`[Int]`, un tipo `[Char]`, `[[String]]`, etc. pero no puede haber un valor cuyo tipo sea simplemente `[]`.

Vamos a jugar un poco con el tipo `Maybe`.

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num t) => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
```

```
ghci> Just 10 :: Maybe Double
Just 10.0
```

Los parámetros de tipo son útiles ya que nos permiten crear diferentes tipos dependiendo del tipo que queramos almacenar en nuestros tipos de datos (valga la redundancia). Cuando hacemos `:t Just "Haha"` el motor de inferencia de tipos deduce que el tipo debe ser `Maybe [Char]`, ya que la `a` en `Justa` es una cadena, luego el `a` en `Maybe a` debe ser también una cadena.

Como habrás visto el tipo de `Nothing` es `Maybe a`. Su tipo es polimórfico. Si una función requiere un `Maybe Int` como parámetro le podemos pasar un `Nothing` ya que no contiene ningún valor. El tipo `Maybea` puede comportarse como un `Maybe Int`, de la misma forma que `5` puede comportarse como un `Int` o como un `Double`. De forma similar el tipo de las listas vacías es `[a]`. Una lista vacía puede comportarse como cualquier otra lista. Por eso podemos hacer cosas como `[1,2,3] ++ []` y `["ha","ha","ha"] ++ []`.

El uso de parámetros de tipo nos puede beneficiar, pero solo en los casos que tenga sentido. Normalmente los utilizamos cuando nuestro tipo de dato funcionará igual sin importar el tipo de dato que contenga, justo como nuestro`Maybe a`. Si nuestro tipo es como una especie de caja, es un buen lugar para usar los parámetros de tipo. Podríamos cambiar nuestro tipo `Car` de:

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

A:

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
```

Pero ¿Tiene algún beneficio? La respuesta es: probablemente no, ya que al final acabaremos escribiendo funciones que solo funcionen con el tipo `CarString String Int`. Por ejemplo, dada la primera definición de `Car`, podríamos crear una función que mostrara las propiedades de un coche con un pequeño texto:

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made
```

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

¡Una función muy bonita! La declaración de tipo es simple y funciona perfectamente. Ahora ¿Cómo sería si `Car` fuera en realidad `Car a b c`?

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) = "This " ++ c ++ " " ++ m ++ " was made
```

Tenemos que forzar a que la función tome un `Car` del tipo `(Show a) => CarString String a`. Podemos ver como la definición de tipo es mucho más complicada y el único beneficio que hemos obtenido es que podamos usar cualquier tipo que sea una instancia de la clase de tipos `Show` como parámetro `c`.

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
```

```
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

A la hora de la verdad, acabaríamos utilizando `Car String String Int` la mayor parte del tiempo y nos daríamos cuenta de que parametrizar el tipo`Car` realmente no importa. Normalmente utilizamos los parámetros de tipo cuando el tipo que está contenido dentro del tipo de dato no es realmente importante a la hora de trabajar con éste. Una lista de cosas es una lista de cosas y no importa que sean esas cosas, funcionará igual. Si queremos sumar una lista de números, mas tarde podemos especificar en la propia función de suma de que queremos específicamente una lista de números. Lo mismo pasa con`Maybe`. `Maybe` representa la opción de tener o no tener un valor. Realmente no importa de que tipo sea ese valor.

Otro ejemplo de un tipo parametrizado que ya conocemos es el tipo `Map k v`de `Data.Map`. `k` es el tipo para las claves del diccionario mientras que`v` es el tipo de los valores. Este es un buen ejemplo en donde los parámetros de tipo son útiles. Al tener los diccionarios parametrizados nos permiten asociar cualquier tipo con cualquier otro tipo, siempre que la clave del tipo sea de la clase de tipos `Ord`. Si estuviéramos definiendo el tipo diccionario podríamos añadir una restricción de clase en la definición:

```
data (Ord k) => Map k v = ...
```

Sin embargo, existe un consenso en el mundo Haskell de que **nunca debemos añadir restricciones de clase a las definiciones de tipo**. ¿Por qué? Bueno, porque no nos beneficia mucho, pero al final acabamos escribiendo más restricciones de clase, incluso aunque no las necesitemos. Si ponemos o no podemos la restricción de clase `Ord k` en la definición de tipo de `Map kv`, tendremos que poner de todas formas la restricción de clase en las funciones que asuman que las claves son ordenables. Pero si no ponemos la restricción en la definición de tipo, no tenemos que poner `(Ord k) =>` en la declaración de tipo de las funciones que no les importe si la clave puede es ordenable o no. Un ejemplo de esto sería la función `toList` que simplemente convierte un diccionario en una lista de asociación. Su declaración de tipo es `toList :: Map k a -> [(k, a)]`. Si `Map k v`tuviera una restricción en su declaración, el tipo de `toList` debería haber sido `toList :: (Ord k) => Map k a -> [(k, a)]` aunque la función no necesite comparar ninguna clave.

Así que no pongas restricciones de clase en las declaraciones de tipos aunque tenga sentido, ya que al final las vas a tener que poner de todas formas en las declaraciones de tipo de las funciones.

Vamos a implementar un tipo para vectores 3D y crear algunas operaciones con ellos. Vamos a usar un tipo parametrizado ya que, aunque normalmente contendrá números, queremos que soporte varios tipos de ellos.

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: (Num t) => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

`vplus` sirve para sumar dos vectores. Los vectores son sumados simplemente sumando sus correspondientes componentes. `scalarMult` calcula el producto escalar de dos vectores y `vectMult` calcula el producto de un vector y un escalar. Estas funciones pueden operar con tipos como `Vector Int`,`Vector Integer`, `Vector Float` o cualquier otra cosa mientras `a`

de`Vector a` sea miembro de clase de tipos `Num`. También, si miras la declaración de tipo de estas funciones, veras que solo pueden operar con vectores del mismo tipo y los números involucrados (como en `vectMult`) también deben ser del mismo tipo que el que contengan los vectores. Fíjate en que no hemos puesto una restricción de clase `Num` en la declaración del tipo `Vector`, ya que deberíamos haberlo repetido también en las declaraciones de las funciones.

Una vez más, es muy importante distinguir entre constructores de datos y constructores de tipo. Cuando declaramos un tipo de dato, la parte anterior al`=` es el constructor de tipos, mientras que la parte que va después (posiblemente separado por `|`) son los constructores de datos. Dar a una función el tipo `Vector t t t -> Vector t t t -> t` sería incorrecto ya que hemos usado tipos en la declaración y el constructor de tipos vector toma un solo parámetro, mientras que el constructor de datos toma tres. Vamos a jugar un poco con los vectores:

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

Instancias derivadas

En la sección *Clases de tipos paso a paso (1ª parte)*, explicamos las bases de las clases de tipo. Dijimos que una clase de tipos es una especie de interfaz que define un comportamiento. Un tipo puede ser una **instancia** de esa clase si soporta ese comportamiento. Ejemplo: El tipo `Int` es una instancia de la clase`Eq`, ya que la clase de tipos `Eq` define el comportamiento de cosas que se pueden equiparar. Y como los enteros se pueden equiparar, `Int` es parte de la clase `Eq`. La utilidad real está en las funciones que actúan como interfaz de `Eq`, que son `==` y `/=`. Si un tipo forma parte de la clase `Eq`, podemos usar las funciones como `==` con valores de ese tipo. Por este motivo, expresiones como `4 == 4` y `"foo" /= "bar"` son correctas.

Mencionamos también que las clases de tipos suelen ser confundidas con las clases de lenguajes como Java, Python, C++ y demás, cosa que más tarde desconcierta a la gente. En estos lenguajes, las clases son como un modelo del cual podemos crear objetos que contienen un estado y pueden hacer realizar algunas acciones. Las clases de tipos son más bien como las interfaces. No creamos instancias a partir de las interfaces. En su lugar, primero creamos nuestro tipo de dato y luego pensamos como qué puede comportarse. Si puede comportarse como algo que puede ser equiparado, hacemos que sea miembro de la clase `Eq`. Si puede ser puesto en algún orden, hacemos que sea miembro de la clase `Ord`.

Más adelante veremos como podemos hacer manualmente que nuestros tipos sean una instancia de una clase de tipos implementando las funciones que esta define. Pero ahora, vamos a ver como Haskell puede automáticamente hacer que nuestros tipos pertenezcan a una de las siguientes clases: `Eq`,`Ord`, `Enum`, `Bounded`, `Show` y `Read`. Haskell puede derivar el comportamiento de nuestros tipos en estos contextos si usamos la palabra clave `deriving` cuando los definimos.

Considera el siguiente tipo de dato:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Describe a una persona. Vamos a asumir que ninguna persona tiene la misma combinación de nombre, apellido y edad. Ahora, si tenemos registradas a dos personas ¿Tiene sentido saber si estos dos registros pertenecen a la misma persona? Parece que sí. Podemos compararlos por igualdad y ver si son iguales o no. Por esta razón tiene sentido que este tipo se miembro de la clase de tipo `Eq`. Derivamos la instancia:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

Cuando derivamos una instancia de `Eq` para un tipo y luego intentamos comparar dos valores de ese tipo usando `==` o `/=`, Haskell comprobará si los constructores de tipo coinciden (aunque aquí solo hay un constructor de tipo) y luego comprobará si todos los campos de ese constructor coinciden utilizando el operador `=` para cada par de campos. Solo tenemos que tener en cuenta una cosa, todos los campos del tipo deben ser también miembros de la clase de tipos `Eq`. Como `String` y `Int` ya son miembros, no hay ningún problema. Vamos a comprobar nuestra instancia `Eq`.

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> let adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41}
ghci> let mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person {firstName = "Michael", lastName = "Diamond", age = 43}
True
```

Como ahora `Person` forma parte de la clase `Eq`, podemos utilizarlo como`a` en las funciones que tengan una restricción de clase del tipo `Eq a`en su declaración, como `elem`.

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

Las clases de tipos `Show` y `Read` son para cosas que pueden ser convertidas a o desde cadenas, respectivamente. Como pasaba con `Eq`, si un constructor de tipos tiene campos, su tipo debe ser miembro de la clase` `Show` o `Read` si queremos que también forme parte de estas clases.

Vamos a hacer que nuestro tipo de dato `Person` forme parte también de las clases `Show` y `Read`.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Ahora podemos mostrar una persona por la terminal.

```
ghci> let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"
```

Si hubiésemos intentado mostrar en la terminal una persona antes de hacer que el tipo `Person` formara parte de la clase `Show`, Haskell se hubiera quejado, diciéndonos que no sabe como representar una persona con una cadena. Pero ahora que hemos derivado la clase `Show` ya sabe como hacerlo.

`Read` es prácticamente la clase inversa de `Show`. `Show` sirve para convertir nuestro tipo a una cadena, `Read` sirve para convertir una cadena a nuestro tipo. Aunque recuerda que cuando uses la función `read` hay que utilizar una anotación de tipo explícita para decirle a Haskell que tipo queremos como resultado. Si no ponemos el tipo que queremos como resultado explícitamente, Haskell no sabrá que tipo queremos.

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

No hace falta utilizar una anotación de tipo explícita en caso de que usemos el resultado de la función `read` de forma que Haskell pueda inferir el tipo.

```
ghci> read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" == mikeD
True
```

También podemos leer tipos parametrizados, pero tenemos que especificar todos los parámetros del tipo. Así que no podemos hacer`read "Just 't'" :: Maybe a` pero si podemos hacer `read "Just 't'" ::Maybe Char`.

Podemos derivar instancias para la clase de tipos `Ord`, la cual es para tipos cuyos valores puedan ser ordenados. Si comparamos dos valores del mismo tipo que fueron definidos usando diferentes constructores, el valor cuyo constructor fuera definido primero es considerado menor que el otro. Por ejemplo, el tipo `Bool` puede tener valores `False` o `True`. Con el objetivo de ver como se comporta cuando es comparado, podemos pensar que está implementado de esta forma:

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

Como el valor `False` está definido primero y el valor `True` está definido después, podemos considerar que `True` es mayor que `False`.

ghci> True *compare* False GT ghci> True > False True ghci> True < False False

En el tipo `Maybe a`, el constructor de datos `Nothing` esta definido antes que el constructor `Just`, así que un valor `Nothing` es siempre más pequeño que cualquier valor `Just algo`, incluso si ese algo es menos un billon de trillones. Pero si comparamos dos valores `Just`, entonces se compara lo que hay dentro de él.

```
ghci> Nothing < Just 100
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

No podemos hacer algo como `Just (*3) > Just (*2)`, ya que `(*3)` y`(*2)` son funciones, las cuales no tienen definida una instancia de`Ord`.

Podemos usar fácilmente los tipos de dato algebraicos para crear enumeraciones, y las clases de tipos `Enum` y `Bounded` nos ayudarán a ello. Considera el siguiente tipo de dato:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Como ningún contructor de datos tiene parámetros, podemos hacerlo miembro de la clase de tipos `Enum`. La clase `Enum` son para cosas que tinen un predecesor y sucesor. Tambien podemos hacerlo miembro de la clase de tipos`Bounded`, que es para

cosas que tengan un valor mínimo posible y valor máximo posible. Ya que nos ponemos, vamos a hacer que este tipo tenga una instancia para todas las clases de tipos derivables que hemos visto y veremos que podemos hacer con él.

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
           deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Como es parte de las clases de tipos `Show` y `Read`, podemos convertir valores de est tipo a y desde cadenas.

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

Como es parte de las clases de tipos `Eq` y `Ord`, podemos comparar o equiparar días.

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

También forma parte de `Bounded`, así que podemos obtener el día mas bajo o el día más alto.

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

También es una instancia de la clase `Enum`. Podemos obtener el predecesor y el sucesor de un día e incluso podemos crear listas de rangos con ellos.

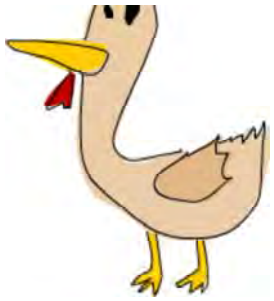
```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

Bastante impresionante.

Sinónimos de tipo

Anteriormente mencionamos que los tipos `[Char]` y `String` eran equivalentes e intercambiables. Esto está implementado con los **sinónimos de tipo**. Los sinónimos de tipo no hacen nada por si solo, simplemente dan a algún tipo un nombre diferente, de forma que obtenga algún significado para alguien que está leyendo nuestro código o documentación. Aquí tienes como define la librería estándar `String` como sinónimo de `[Char]`.

```
type String = [Char]
```



Acabamos de intrudir la palabra clave `type`. Esta palabra clave podría inducir a errores a algunos, ya que en realidad no estamos haciendo haciendo nada nuevo (lo hacemos con la palabra clave `data`). Simplemente estamos dando un sinónimos a un tipo que ya existe.

Si hacemos una función que convierta una cadena a mayúscuals y la llamamos`toUpperString` o algo parecido, podemos darle una declaración de tipo como`toUpperString :: [Char] -> [Char]` o `toUpperString :: String -> String`. Ambas son esecialmente lo mismo, solo que la última es más legible.

Cuando estabamos hablando del módulo `Data.Map`, primero presentamos una agenda de teléfonos representada con una lista de asociación para luego convertirla en un diccionario. Como ya sabemos, una lista de asociación no es más que una lista de duplas clave-valor. Vamos a volver a ver la lista que teníamos.

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

Vemos que el tipo de `phoneBook` es `[(String,String)]`. Esto nos dice que es una lista de asociación que asocia cadenas con cadena, pero nada más. Vamos a crear un sinónimo de tipo para transmitir algo más de información en la declaración de tipo.

```
type PhoneBook = [(String,String)]
```

Ahora la declaración de tipo de nuestra función `phoneBook` sería`phoneBook :: PhoneBook`. Vamos a hacer un sinónimo de tipo para las cadenas también.

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Dar un sinónimo al tipo `String` es algo que suelen hacer los programadores de Haskell cuando quieren transmitir algo más de información acerca del cometido de las cadenas en sus funciones y que representan.

Así que ahora, cuando implementemos una función que tome el nombre y el número de teléfono y busque si esa combinación está en nuestra agenda telefónica, podremos darle una declaración de tipo muy descriptiva:

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

Si decidimo no utilizar sinónimos de tipo, nuestra función tendría la declaración de tipo `String -> String -> [(String,String)] -> Bool`. En este caso, la declaración de tipo que utiliza los sinónimos de tipo es mucho más clara y fácil de entender. Sin embargo, no debes abusar de ellos. Utilizamos los sinónimos de tipo o bien para indicar que representa un tipo que ya existe en nuestras funciones (y de esta forma nuestras delcaraciones de tipo se convierten en la mejor documentación) o bien cuando algo tiene un tipo muy largo que se repite mucho (como `[(String,String)]`) y tiene un significado concreto para nosotros.

Los sinónimos de tipo también pueden ser parametrizados. Si queremos un tipo que represente las listas de asociación pero también queremos que sea lo suficientemente general como para utilizar cualquier tipo de clave y valor, podemos utilizar esto:

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

Con esto, una función que tomara un valor por clave en una lista de asociación puede tener el tipo `(Eq k) => k -> AssocList k v -> Maybe v`.`AssocList` es un constructor de tipos que toma dos tipos y produce un tipo concreto, como `AssocList Int String` por ejemplo.

Nota
Cuando hablamos de tipos concretos nos referimos a tipos completamente aplicados, como `Map Int String`. A veces, los chicos y yo decimos que `Maybe` es un tipo, pero no queremos referirnos a eso, ya que cualquier idiota sabe que `Maybe` es un constructor de tipos. Cuando aplico un tipo extra a `Maybe`, como`Maybe String`, entonces tengo un tipo concreto. Ya sabes, los valores solo pueden tener tipos que sean tipos concretos. Concluyendo, vive rápido, quiere mucho y no dejes que nadie te tome el pelo.

De la misma forma que podemos aplicar parcialmente funciones para obtener nuevas funciones, podemos aplicar parcialmente los parámetros de tipo y obtener nuevos constructores de tipo. De la misma forma que llamamos a la funciones con parámetros de menos para obtener nuevas funciones, podemos especificar un constructor de tipos con parámetros de menos y obtener un constructor de tipos parcialmente aplicado. Si queremos un tipo que represente un diccionario (de `Data.Map`) que asocie enteros con cualquier otra cosa, podemos utilizar esto:

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

O bien esto otro:

```
type IntMap = Map Int
```

De cualquier forma, el constructor de tipos `IntMap` tomará un parámetro y ese será el tipo con el que se asociarán los enteros.

Nota
Si vas a intentar implementar esto, seguramente imporatarás de forma cualificada el módulo `Data.Map`. Cuando realizas una importación cualificada, los constructores de tipo también deben estar precedidos con el nombre del módulo. Así que tienes que escribir algo como `type IntMap = Map.Map Int`.

Asegurate de que realmente entiendes la diferencia entre constructores de tipos y constructores de datos. Solo porque hayamos creado un sinónimo llamado`IntMap` o `AssocList` no significa que podamos hacer cosas como`AssocList [(1,2),(4,5),(7,9)]`. Lo único que significa es que podemos referirnos a ese tipo usando nombres diferentes. Podemos hacer `[(1,2),(3,5),(8,9)] :: AssocList Int Int`, lo cual hará que los número de adentro asuman el tipo `Int`, pero podemos seguir usando esta lista como si fuera una lista que albergara duplas de enteros. Lo sinónimos de tipo (y los tipos en general) solo pueden ser utlizados en la porción de Haskell dedicada a los tipos. Estaremos en esta porción de Haskell cuando estemos definiendo tipos nuevos (tanto en las declaraciones `data` como en las de`type`) o cuando nos situemos después de un `::`. `::` se utiliza solo para las declaraciones o anotaciones de tipo.

Otro tipo de dato interesante que toma dos tipos como parámetro es el tipo`Either a b`. Así es como se define más o menos:

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

Tiene dos constructores de datos. Si se utiliza `Left`, entonces contiene datos del tipo `a` y si se utiliza `Right` contiene datos del tipo `b`. Podemos utilizar este tipo para encapsular un valor de un tipo u otro y así obtener un valor del tipo `Either a b`. Normalmente utilizaremos un ajuste de patrones con ambos, `Left` y `Right`, y nos diferenciaremos según sea uno u otro.

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

Hasta ahora hemos visto que `Maybe a` es utilizado para representar resultados de cálculos que podrían haber fallado o no. Pero a veces,`Maybe a` no es suficientemente bueno ya que `Nothing` únicamente nos informa de que algo ha fallado. Esto esta bien para funciones que solo pueden fallar de una forma o si no nos interesa saber porque y como han fallado. Una búqueda en un `Data.Map` solo falla cuando la clave que estamos buscando no se encuentra en el diccionario, así que sabemos exacmente que ha pasado. Sin embargo, cuando estamos interesados en el cómo o el porqué a fallado algo, solemos utilizar como resultado el tipo `Either a b`, donde `a` es alguna especie de tipo que pueda decirnos algo sobre un posible fallo, y `b` es el tipo de un cálculo satisfactorio. Por lo tanto, los errores usan el constructor de datos `Left` mientras que los resultado usan `Right`.

Un ejemplo: un instituto posee taquillas para que sus estudiantes tengan un lugar donde guardar sus posters de *Guns'n'Roses*. Cada taquilla tiene una combinación. Cuando un estudiante quiere una taquilla nueva, le dice al supervisor de las taquillas que número de taquilla quiere y él le da un código para esa taquilla. Sin embargo, si alguien ya está usando la taquilla, no le puede decir el código y tienen que elegir una taquilla diferente. Utilizaremos un diccionario de `Data.Map` para representar las taquillas. Asociará el número de la taquilla con duplas que contengan si la taquilla está en uso o no y el código de la taquilla.

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

Bastante simple. Hemo creado un nuevo tipo de dato para representar si una taquilla está libre o no, y hemos creado un sinónimo para representar el código de una taquilla. También creado otro sinónimo para el tipo que asocia los los números de las taquillas con las duplas de estado y código. Ahora, vamos a hacer una función que busque un número de taquilla en el diccionario. Vamos a usar el tipo `Either String Code` para representar el resultado, ya que nuestra búsqueda puede fallar de dos formas: la taquilla ya ha sido tomada, en cuyo caso decimos quien la posee o si el no hay ninguna taquilla con ese número. Si la búqueda falla, vamos a utilizar una cadena para obtener el porqué.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing -> Left $ "Locker number " ++ show lockerNumber ++ " doesn't exist!"
        Just (state, code) -> if state /= Taken
                              then Right code
                              else Left $ "Locker " ++ show lockerNumber ++ " is alread
```

Hacemos una búsqueda normal en un diccionario. Si obtenemos `Nothing`, devolvemos un valor con el tipo `Left String` que diga que esa taquilla no existe. Si la encontramos, hacemos una comprobación adicional para ver si la taquilla está libre. Si no lo está, devolvemos un `Left` diciendo que la taquilla a sido tomada. Si lo está, devolvemos un valor del tipo `RightCode`, el

cual daremos al estudiante. En realidad es un `Right String`, aunque hemos creado un sinónimo para añadir un poco más de información en la declaración de tipo. Aquí tienes un diccionario de ejemplo:

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken,"ZD39I"))
    ,(101,(Free,"JAH3I"))
    ,(103,(Free,"IQSA9"))
    ,(105,(Free,"QOTSA"))
    ,(109,(Taken,"893JJ"))
    ,(110,(Taken,"99292"))
    ]
```

Vamos a buscar el código de unas cuantas taquillas:

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

Podríamos haber utlizado el tipo `Maybe a` para representar el resultado pero entonces no sabríamos el motivo por el cual no podemos obtener el código. Ahora, tenemos información acerca del fallo en nuestro tipo del resultado.

Estructuras de datos recursivas

Como ya hemos visto, un costructor de un tipo de dato algebraico puede tener (o no tener) varios campos y cada uno de estos debe ser un tipo concreto. Teniendo esto en cuenta, podemos crear tipos cuyos campos de constructor sean el propio tipo. De esta forma, podemos crear estructuras de datos recursivas, en el que un valor de un cierto tipo contenga valores de ese mismo tipo, el cual seguirá conteniendo valores del mismo tipo y así sucesivamente.

Piensa en la lista `[5]`. Es lo mismo que `5:[]`. A la izquierda del `:`hay un valore, y a la derecha hay una lista. En este caso, una lista vacía. ¿Qué pasaría con la lista `[4,5]`? Bueno, es lo mismo que `4:(5:[])`. Si miramos el primer `:`, vemos que también tiene un elemento a su izquierda y una lista a su derecha `(5:[])`. Lo mismo sucede para la lista`3:(4:(5:6:[]))`, que también podría escribirse como `3:4:5:6:[]` (ya que`:` es asociativo por la derecha) o `[3,4,5,6]`.

Podemos decir que una lista es o bien una lista vacia o bien un elemento unido con un `:` a otra lista (que puede ser una lista vacía o no).

¡Vamos a usar los tipod de datos algebraicos para implementar nuestra propia lista!

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

Se lee de la misma forma que se leía nuestra definición de lista en un párrafo anterior. Es o bien una lista vacía o bien una combinación de un elemento y otra lista. Si estás confundido con esto, quizás te sea más fácil entenderlo con la sintaxis de registro:

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

Puede que también estes confundido con el constructor `Cons`. `Cons` es otra forma de decir `:`. En realidad, en las listas, `:` es un constructor que toma un valor y otra lista y devuleve una lista. En otras palabras, tiene dos campos. Uno es del tipo `a` y otro es del tipo `[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))
```

Si hubiésemos llamado a nuestro constructor de forma infija podrías ver mejor como es simplemente `:`. `Empty` es como `[]` y `4 `Cons` (5 `Cons` Empty)` es como `4:(5:[])`.

Podemos definir funciones que automáticamente sean infijas si las nombramos únicamente con caracteres especiales. Podemos hacer lo mismo con los constructores, ya que son simplemente funciones que devuelve un tipo de dato concreto. Mira esto:

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

Antes de nada, vemos que hay una nueva construcción sintáctica, una declaración infija. Cuando definimos funciones como operadores, podemos usar esta cosntrucción para darles un determinado comportamiento (aunque no estamos obligados a hacerlo). De esta forma definimos el orden de precedencia de un operador y si asociativo por la izquierda o por la derecha. Por ejemplo, `*`es `infixl 7 *` y `+` es `infixl 6 +`. Esto siginifica que ambos son asociativos por la izquierda de forma que `(4 * 3 * 2)` es `(4 * 3) * 2)`pero `*` tiene un orden de precedencia mayor que `+`, por lo que`5 * 4 + 3` es equivalente a `(5 * 4) + 3`.

De qualquier modo, al final acabamos escribiendo `a :-: (List a)` en lugar de `` Cons a (List a)``. Ahora podemos escribir las listas así:

```
ghci> 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Empty)))
```

Haskell serguirá mostrando el cosntructor como una función prefija cuando derivemos `Show`, por este motivo aparecen los poréntesis alrededor del constructor (recuerda que `4 + 3` es igual que `(+) 4 3`).

Vamos a crear una función que una dos de nuestras listas. Así es como está definida la función `++` para listas normales:

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

Así que copiamos esta definición y la aplicamos a nuestras listas:

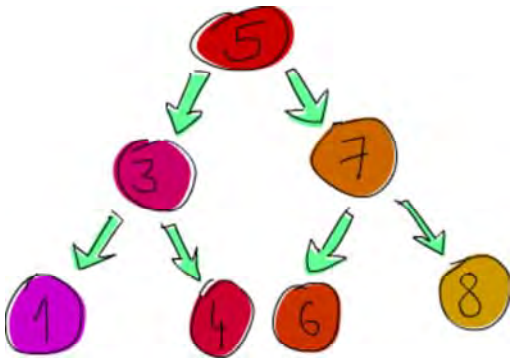
```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

Y así es como funciona:

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

Bien. Si te apetece puedes implementar todas las funciones que operan con listas con nuestro tipo de listas.

Fíjate que hemos utilizado un ajuste de patrón `(x :-: xs)`. Esto función ya que el ajuste de patrones en realidad funciona ajustando constructores. Podemos ajustar un patrón `:-:` porque es un constructor de nuesto tipo de la misma forma que `:` es un constructor de las listas estándar. Lo mismo sucede para `[]`. Ya que el ajuste de patrones funciona (solo) con constructores de datos, podemos ajustar patrones como los constructores prefijos normales, constructores infijos o cosas como `8` o `'a'`, que al fin y al cabo son constructores de tipos númericos y caracteres.



Vamos a implementar un árbol binario de búsqueda. Si no estás familiarizado con los árboles binarios de búsqueda de otros lenguajes como *C*, aquí tienes una expliación de lo que son: un elemento apunta a otros dos elementeos, uno esta a la izquierda y otro a la derecha. El elemento a la izquierda es más pequeño y el segundo es más grande. Cada uno de estos dos elementos puede apuntar a otros dos elementos (o a uno o a ninguno). En efecto, cada elemento tienen sus propios sub-árboles. Lo bueno de los árboles binarios de búsqueda es que sabemos que todos los elementos que están en el sub-árbol de la iquierda de, 5, por ejemplo, son menores que 5. Lo elementos que están en el sub-árbol de la derecha son mayores. Así que si estamos buscando el elemento 8 en nuestro árbol, emepezamos comparándolo con 5, como vemos que es menor que 5, nos vamos al sub-árbol de la derecha. Ahora estaríamos en 7, como es menor que 8 continuaríamos hacia la derecha. De esta formá encontraríamos el elemento en tres pasos. Si estuvieramos usando una lista (o un árbol no balanceado), nos hubiera costado unos 7 pasos encontrar el 8.

Los conjuntos y diccionario de `Data.Set` y `Data.Map` están implementandos utilizando árboles, solo que en lugar de árboles binarios de búsqueda, utilizan árboles binarios de búsqueda balanceados, de forma que estén siempre balanceados. Ahora implementaremos simplemente árboles binarios de búsqueda normales.

Vamos a decir que: un árbol es o bien un árbol vacío o bien un elemento que contiene un elemento y otros dos árboles. Tiene pinta de que va a encajar perfectamente con los tipos de datos algebraicos.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

Vale. En lugar de construir manualmente un árbol, vamos a crear una función que tome un elemento y un árbol e inserte dicho elemento en su posición adecuada dentro del árbol. Hacemos esto comparando el elemento que queremos insertar con la raíz del árbol y si es menor, vamos a la izquierda y si no a la derecha. Hacemos lo mismo para coda nodo siguiente hasta que alcanzemos un árbol vacío. Cuando lo hagamos simplemente insertamos el elmento en lugar del árbol vacío.

En lenguajes como *C*, realizamos esta tarea modificando los punteros y valores del árbol. En Haskell, no podemos modificar nuestro árboles, así que tenemos que crear un nuevo sub-árbol cada vez que decidamos si vamos a la derecha o a la izquierda y al final la función de inserción devolver un árbol complentamente nuevo, ya que Haskell no tiene el concepto de puntero. Así pues la declaración de tipo de nuestra función será alfgo como `a ->Tree a - > Tree a`. Toma un elemento y un árbol y devuelve un nuevo árbol que posee en su interior dicho elemento. Puede parecer ineficienciente pero la evaluación perezosa de Hasekell ya se encarga de ello.

Aqui tienes dos funciones. Una de ellas es una función auxiliar para crear un árbol unitario (que solo contiene un elemento) y la otra es una función que inserta elementos en un árbol.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

La función `singleton` es forma rápida de crear un árbol que contenga un elemento y dos sub-árboles vacios. En la función de inserción, tenemos como primer patrón el caso base. Si hemos alcanzado un sub-árbol vacio, esto significa que estamos donde queríamos y en lugar de un árbol vacío, queremos un árbol unitario que contenga el elemento a insertar. Si no estamos insertando el elemento en un árbol vacío tenemos que comprobar varias cosas. Primero, si el elemento que vamos a insertar es el mismo que la raíz del sub-árbol, simplemente devolvemos el árbol como estaba. Si es menor, devolvemos un árbol que tenga la misma raíz, el mimso sub-árbol derecho pero en lugar de su sub-árbol izquierdo, ponemos el árbol que va a contener dicho elemento. Lo mismo ocurre (pero en sentido contrario) para los valores que son mayores que el elemento raíz.

A continuación vamos a crear una función que compruebe si un elemento pertence a un árbol. Primero vamos a definir el caso base. Si estamos buscando un elemento en un árbol vacío, obviamente el elemento no está ahí. Vale, fíjate que esto es básicamente lo mismo que el caso base de la búsqueda en listas: si estamos buscando un elemento en una lista vacía, obviamente el elemento no está ahí. De todos modos, si no estamos buscando el elemento en un árbol vacío, entonces tenemos que hacer varias comprobaciones. Si el elemento que estamos buscando es el elemento raíz ¡Genial! ¿Y si no lo es? Bueno, tenemos la ventaja de que sabemos que todos los elementos menores que la raíz están en el sub-árbol izquierdo. Así que si el elemento que estamos buscando es menor que la raiz, comprobamos si el elemento está en el sub-árbol izquierdo. Si es mayor, comprobamos el sub-árbol derecho.

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right
```

¡Vamos a divertirnos con nuestro árboles! En lugar de contruir manualmente un árbol (aunque podríamos), usaremos un pliegue para construir un árbol a partir de una lista. Recuerda, casi cualquier cosa que recorra una lista elemento a elemento y devuelve alguna especie de valor puede ser implementado con un pliegue. Empezaremos con un árbol vacío y luego recorreremos la lista desde la derecha e iremos insertando elementos a nuestro árbol acumulador.

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 4 EmptyTree EmptyTree)) (Node 7 (Node 6
```

En este `foldr`, `treeInsert` es la función de pliegue (toma un árbol y un elemento de la lista y produce un nuevo árbol) y `EmptyTree` es el valor inicial. Por supuesto, `nums` es la lista que estamos plegando.

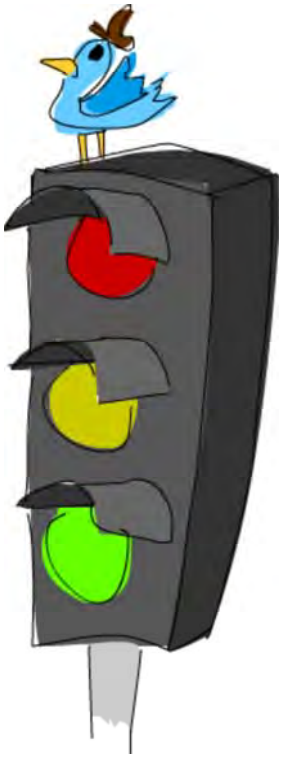
No es muy legible el árbol que se muestra por la consola, pero si lo intentamos, podemos descifrar su estructura. Vemos que el nodo raíz es 5 y luego tiene dos sub-árboles, uno que tiene como elemento raíz a 3, y otro a 7.

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

Vamos que comprobar la pertencia de un elemento a un árbol funciona perfectamente. Genial.

Como puede ver los tipos de datos algebraicos en Haskell son un concepto muy intersante a la vez que pontentes. Podemos utilizarlos desde para representar valores booleanos hasta enumeraciónes de los días de la semana, e incluso árboles binarios de búsquedas.

Clases de tipos paso a paso (2ª parte)



Hasta ahora hemos aprendido a utilizar algunas clases de tipos estándar de Haskell y hemos visto que tipos son miembros de ellas. También hemos aprendido a crear automáticamente instancias de nuestros tipos para las clases de tipos estándar, pidiéndole a Haskell que las derive por nostros. En esta sección vamos a ver como podemos crear nuestras propias clases de tipo y a como crear instancias de tipos para ellas a mano.

Un pequeño recordatorio acerca de las clases de tipos: las clases de tipos son como las interfaces. Una clase de tipos define un comportamiento (como comparar por igualdad, comparar por orden, una enumeración, etc.) y luego ciertos tipos pueden comportarse de forma a la instancia de esa clase de tipos. El comportamiento de una clase de tipos se consigue definiendo funciones o simplemente definiendo tipos que luego implementaremos. Así que cuando digamos que un tipo es una instancia de un clase de tipos, estamos diciendo que podemos usar las funciones de esa clase de tipos con ese tipo.

Las clases de tipos no tienen nada que ver con las clases de *Java* o *Pyhton*. Esto suele confundir a mucha gente, así que me gustaría que olvidaras ahora mismo todo lo que sabes sobre las clases en los lenguajes imperativos.

Por ejemplo, la clase de tipos `Eq` es para cosas que pueden ser equiparadas. Define las funciones `==` y `/=`. Si tenemos un tipo (digamos, `Car`) y el comparar dos coches con la función `==` tiene sentido, entonces tiene sentido que `Car` sea una instancia de `Eq`.

Así es como está defina la clase `Eq` en `Prelude`:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

¡Alto, alto, atlo! ¡Hay mucha sintaxis y palabras raras ahí! No te preocupes, estará todo claro en un segundo. Lo primero de todo, cuando escribimos`class Eq a where` significa que estamos definiendo una clase de tipos nueva y que se va a llamar `Eq`. La `a` es la variable de tipo y significa que`a` representará el tipo que dentro de poco hagamos instancia de `Eq`. No tiene porque llamarse `a`, de hecho no tiene ni que ser de una sola letra, solo debe ser una palabra en minúsculas. Luego definimos varias funciones. No es obligatorio implementar los cuerpos de las funciones, solo debemos especificar las declaraciones de tipo de las funciones.

Nota

Hay gente que entederá esto mejor si escribimos algo como`class Eq equiparable where` y luego definimos el tipo de las funciones como `(==) :: equiparable -> equiparable -> Bool`.

De todos modos, hemos implementado el cuerpo de las funciones que define`Eq`, solo que las hemos implementado en terminos de recursión mutua. Decimos que dos instancias de la clase `Eq` son iguales si no son desiguales y son desiguales y no son iguales. En realidad no teníamos porque haberlo echo, pero pronto veremos de que forma nos ayuda.

Nota

Si tenemos un `class Eq a where` y definimos una declaración de tipo dentro de la clase como `(==) :: a -> -a -> Bool`, luego, cuando examinemos el tipo de esa función obtendremos`(Eq a) => a -> a -> Bool`.

Así que ya tenemos una clase ¿Qué podemos hacer con ella? Bueno, no mucho. Pero una vez empezemos a declarar instancias para esa clase, empezaremos a obtener algun funcionalidad útil. Mira este tipo:

```
data TrafficLight = Red | Yellow | Green
```

Define los estados de un semáforo. Fijate que no hemos derivado ninguna instancia, ya que vamos a escribirlas a mano, aunque podríamos haberlas derivado para las clases `Eq` y `Show`. Aquí tienes como creamos la instancia para la clase `Eq`.

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

Lo hicimos utilizando la palabra clave `instance`. Así que `class` es para definir nuevas clases de tipos y `instance` para hacer que nuestros tipos tengan una instancia para cierta clase de tipos. Cuando estabamos definiendo `Eq` escribimos `class Eq a where` y dijimos que `a`representaría el tipo que hiciéramos instancia después. Lo podemos ver claramente ahora, ya que cuando estamos escribiendo una instancia, escribimos`instance Eq TrafficLight where`. Hemo remplazado la `a` por el tipo actual.

Como `==` fue definido en la definición de clase en términos de `/=` y viceversa, solo tenemos que sobreescribir una de ellas en la delcaración de instancia. A esto se le llama la definición completa mínima de una clase de tipos, o dicho de otra forma, el mínimo número de funciones que tenemos que implementar para que nuestro tipo pertenezca a una determinada clase de tipos. Para rellenar la definición completa mínima de `Eq`, tenemos que sobreescribir o bien `==` o `/=`. Si `Eq` hubiese sido definido como:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Tendríamos que haber implementado ambas funciones a la hora de crear una instancia, ya que Hasekell sabría como están relacionadas esas funciones. De esta forma, la definición completa mínima serían ambas, `==` y `/=`.

Como has visto hemos implementado `==` usando ajuste de patrones. Como hay muchos más casos donde dos semáforos no están en el mismo estado, especificamos para cuales son iguales y luego utilizamos un patrón que se ajuste a cualquier caso que no sea ninguno de los anteriores para decir que no son iguales.

Vamos a crear también una instancia para `Show`. Para satisfacer la definición completa mínima de `Show`, solo tenemos que implementar la función `show`, la cual toma un valor y lo convierte a una cadena.

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

Una vez más hemos utilizado el ajuste de patrones para conseguir nuestros objetivos. Vamos a verlo en acción:

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Perfecto. Podríamos haber derivado `Eq` y hubiera tenido el mismo efecto. Sin embargo, derivar `Show` hubiera representando directamente los constructores como cadenas. Pero si queremos que las luces aparezcan como `"Red light"` tenemos que crear esta instancia a mano.

También podemos crear clases de tipos que sean subclases de otras clases de tipos. La declaración de la clase `Num` es un poco larga, pero aquí tienes el principio:

```
class (Eq a) => Num a where
   ...
```

Como ya hemos mencionado anteriormente, hay un montón de sitios donde podemos poner restriciones de clases. Esto es lo mismo que escribir `class Num awhere`, solo que decimos que nuestro tipo `a` debe ser una instancia de`Eq`. Basicamente decimos que hay que crear la instancia `Eq` de un tipo antes de que éste forme parte forme parte de la clase `Num`. Antes de que un tipo se pueda considerar un número, tiene sentido que podamos determinar si los valores de un tipo puede sen equiparados o no. Esto es todo lo que hay que saber de las subclases ya que simplemente son restriscciones de clase dentro de la definición de una clase. Cuando definamos funciones en la declaración de una clase o en la definición de una instancia, podemos asumir que `a` es parte de la clase `Eq` así que podemos usar `==` con los valores de ese tipo.

¿Pero cómo son creadas las instancias del tipo `Maybe` o de las listas? Lo que hace diferente a `Maybe` de, digamos, `TrafficLight` es que `Maybe`no es por si mismo un tipo concreto, es un constructor de tipos que toma un parámetro (como `Char` o cualquier otra cosa) para producir un tipo concreto. Vamos a echar un vistazo a la clase `Eq` de nuevo:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

A partir de la declaración de tipo, podemos observar que `a` es utilizado como un tipo concreto ya que todos los tipos que aparecer en una función deben deben ser concretos (Recuerda, no puedes tener una función con el tipo`a -> Maybe` pero si una función `a -> Maybe a` o `Maybe Int -> MaybeString`). Por este motivo no podemos hacer cosas como:

```
instance Eq Maybe where
    ...
```

Ya que como hemos visto, `a` debe ser un tipo concreto pero `Maybe` no lo es. Es un constructor de tipos que toma un parámetro y produce un tipo concreto. Sería algo pesado tener que escribir `instance Eq (Maybe Int)`where`, `instance Eq (Maybe Char) where`, etc. para cada tipo. Así que podemos escribirlo así:

```
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

Esto es como decir que queremos hacer una instancia de `Eq` para todos los tipos `Maybe algo`. De hecho, podríamos haber escrito `Maybe algo`, pero preferimos elegir nombres con una sola letra para ser fieles al estilo de Haskell. Aquí, `(Maybe m)` hace el papel de `a` en `class Eq a where`. Mientras que `Maybe` no es un tipo concreto, `Maybe m` sí. Al utilizar un parámetro tipo (`m`, que está en minúsculas), decimos que queremos todos los tipos que sean de la forma `Maybe m`, donde `m` es cualquier tipo que forme parte de la clase `Eq`.

Sin embargo, hay un problema con esto ¿Puedes averiguarlo? Utilizamos `==`sobre los contenidos de `Maybe` pero nadie nos asegura de que lo que contiene `Maybe` forme parte de la clase `Eq`. Por este motivo tenemos que modificar nuestra declaración de instancia:

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

Hemos añadido una restricción de clase. Con esta instancia estamos diciendo: Queremos que todos los tipos con la forma `Maybe m` sean miembros de la clase de tipos `Eq`, pero solo aquellos tipos donde `m` (lo que está contenido dentro de `Maybe`) sean miembros también de `Eq`. En realidad así sería como Haskell derivaría esta instancia.

La mayoría de las veces, las restricciones de clase en las *declaraciones de clases* son utilizadas para crear una clases de tipos que sean subclases de otras clases de tipos mientras que las restricciones de clase en las*declaraciones de instancias* son utilizadas para expresar los requisitos de algún tipo. Por ejemplo, ahora hemos expresado que el contenido de `Maybe` debe formar parte de la clase de tipos `Eq`.

Cuando creas una instancia, si ves que un tipo es utilizado como un tipo concreto en la declaración de tipos (como `a` en `a -> a -> Bool`), debes añadir los parámetros de tipos correspondientes y rodearlo con paréntesis de forma que acabes teniendo un tipo concreto.

Nota
Ten en cuenta que el tipo para el cual estás trantando de hacer una instancia remplazará el parámetro de la declaración de clase. La`a` de `class Eq a where` será remplazada con un tipo real cuando crees una instancia, así que trata mentalmente de poner el tipo en la declaración de tipo de las funiones.`(==) :: Maybe -> Maybe -> Bool` no tiene mucho sentido, pero `(==) :: (Eq m) => Maybe m -> Maybe m -> Boo` sí. Pero esto es simplemente una forma de ver las cosas, ya que `==` simpre tendrá el tipo `(==) :: (Eq a) => a -> a -> Bool`, sin importar las instancias que hagamos.

Oh, una cosa más. Si quieres ver las instancias que existen de una clase de tipos, simplemente haz `:info YourTypeClass` en GHCi. Así que si utilizamos`:info Num` nos mostrará que funciones están definidas en la clase de tipos y nos mostrará también una lista con los tipos que forman parte de esta clase.`:info` también funciona con tipos y constructores de tipo. Si hacemos`:info Maybe` veremos todas las clases de tipos de las que éste forma parte.`:info` también te muestra el tipo de una función. Bastante útil.

La clase de tipos Yes-No

En JavaScript y otros lenguajes débilmente tipados, puedes poner casi cualquier cosa dentro de una expresión. Por ejemplo, puedes hacer todo lo siguiente: `if (0) alert ("YEAH!") else alert("NO!")`, `if ("") alert("YEAH!") else alert("NO!")`, `if (false) alert("YEAH") else alert("NO!)`, etc. Y todos estos mostrarán un mensaje diciendo `NO!`. Si hacemos`if ("WHAT") alert ("YEAH") else alert("NO!")` mostrará `"YEAH!"` ya que en JavaScript las cadenas no vacías son consideradas valores verdaderos.

Aunque el uso estricto de `Bool` para la semántica de booleanos funciona mejor en Haskell, vamos a intentar implementar este comportamiento de JavaScript ¡Solo para divertirnos! Empecemos con la declaración de clase.

```
class YesNo a where
    yesno :: a -> Bool
```

Muy simple. La clase de tipos `YesNo` define una función. Esta función toma un valor de un tipo cualquiera que puede expresar algún valor de verdad y nos dice si es verdadero o no. Fíjate en la forma que usamos `a` en la función,`a` tiene que ser un tipo concreto.

Lo siguiente es definir algunas instancias. Para los números, asumimos que (como en JavaScript) cualquier número que no sea 0 es verdadero y 0 es falso.

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

La listas vacías (y por extensión las cadenas) son valores falsos, mientras que las listas no vacías tienen un valor verdadero.

```
instance YesNo [a] where
    yesno [] = False
    yesno _ = True
```

Fíjate como hemos puesto un parámetro de tipo dentro para hacer de la lista un tipo concreto, aunque no suponemos nada acerca de lo que contiene la lista. Qué más... Mmmm... ¡Ya se! `Bool` también puede contener valores verdaderos y falos y es bastante obvio cual es cual.

```
instance YesNo Bool where
    yesno = id
```

¿Eh? ¿Qué es `id`? Simplemente es una función de la librería estándar que toma un parámetro y devuelve lo mismo, lo cual es lo mismo que tendríamos que escribir aquí.

Vamos a hacer también una instancia para `Maybe a`.

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing = False
```

No necesitamos una restricción de clase ya que no suponemos nada acerca de los contenidos de `Maybe`. Simplemente decimos que es verdadero si es un valor`Just` y falso si es `Nothing`. Seguimos teniendo que escribir`(Maybe a)` en lugar de solo `Maybe` ya que, si lo piensas un poco, una función `Maybe -> Bool` no puede existir (ya que `Maybe` no es un tipo concreto),

mientras que `Maybe a -> Bool` es correcto. Aun así, sigue siendo genial ya que ahora, cualquier tipo `Maybe algo` es parte de la clase`YesNo` y no importa lo que sea `algo`.

Antes definimos un tipo `Tree a` para representar la búsqueda binaria. Podemos decir que un árbol vacío tiene un valor falso mientras cualquier otra cosa tiene un valor verdadero.

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _ = True
```

¿Puede ser el estado de un semáforo un valor verdadero o falso? Claro. Si está rojo, paras. Si está verde, continuas. ¿Si está ámbar? Ehh... normalmente suelo acelerar ya que vivo por y para la adrenalina.

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _ = True
```

Genial, ahora tenemos unas cuantas instancias, vamos a jugar con ellas:

```
hci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: (YesNo a) => a -> Bool
```

Bien ¡Funciona! Vamos a hacer una función que imite el comportamiento de una sentencia `if`, pero que funcione con valores `YesNo`.

```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult = if yesno yesnoVal then yesResult else noResult
```

Bastante simple. Toma un valor con un grado de verdad y otros dos valores más. Si el primer valor es verdadero, devuelve el primer valor de los otros dos, de otro modo, devuelve el segundo.

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

La clase de tipos funtor

Hasta ahora, nos hemos encontrado con un montón de clases de tipos de la librería estándar. Hemos jugado con `Ord`, la cual es para cosas que pueden ser ordenadas. Hemos visto `Eq`, que es para cosas que pueden ser equiparadas. Vimos también `Show`, la cual sirve como interfaz para los tipos cuyos valores pueden ser representados como cadenas. Nuestro buen amigo`Read` estará aquí siempre que necesitemos convertir una cadena a un valor de algún tipo. Y ahora, vamos a echar un vistazo a la clase de tipos`Functor`, la cual es básicamente para cosas que se pueden mapear. Seguramente ahora mismo estés pensando en listas, ya que mapear una lista es algún muy común en Haskell. Y estás en lo cierto, el tipo lista es miembro de la clase de tipos `Functor`.

¿Qué mejor forma de conocer la clase de tipos `Functor` que ver como está implementada? Vamos a echar una ojeada.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

De acuerdo. Hemos visto que define una función, `fmap`, y no proporciona ninguna implementación por defecto para ella. El tipo de `fmap` es interesante. En las definiciones de clases de tipos que hemos visto hasta ahora, la variable de tipo que ha tenido un papel importante en la clase de tipos ha sido un tipo concreto, como `a` en `(==) :: (Eq a) => a -> a ->Bool`. Pero ahora, `f` no es un tipo concreto (un tipo que puede tener un valor, como `Int`, `Bool` o `Maybe String`), sino un constructor de tipos que toma un tipo como parámetro. Un ejemplo rápido para recordar:`Maybe Int` es un tipo concreto, pero `Maybe` es un constructor de tipos que toma un tipo como parámetro. De cualquier modo, hemo visto que `fmap` toma una función de un tipo a otro y un funtor aplicado a un tipo y devuelve otro funtor aplicado con el otor tipo.

Si esto te suena un poco confuso, no te preocupes. Lo verás todo más claro ahora cuando mostremos un cuantos ejemplos. Mmm... esta declaración de tipo me recuerda a algo. Si no sabes cual es el tipo de `map`, es este: `map ::(a -> b) -> [a] -> [b]`.

¡Interesante! Toma una función de un tipo a otro y una lista de un tipo y devuelve una lista del otro tipo. Amigos, creo que acabamos de descubir un funtor. De hecho, `map` es `fmap` pero solo funciona con listas. Aquí tienes como las listas tienen una instancia para la clase `Functor`.

```
instance Functor [] where
    fmap = map
```

¡Eso es! Fíjate que no hemos escrito `instance Functor [a] where`, ya que a partir de `fmap :: (a -> b) -> f a -> f b` vemos que `f` tiene que ser un cosntructor de tipos que toma un parámetro. `[a]` ya es un tipo concreto (un lista con cualquier tipo dentro), mientras que `[]` es un constructor de tipos que toma un parámetro y produce cosas como `[Int]`, `[String]` o incluso `[[String]]`.

Como para las listas, `fmap` es simplemente `map`, obtenemos el mismo resultado cuando las usamos con listas.

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

¿Qué pasa cuando realizamos `map` o `fmap` sobre listas vacías? Bien, desde luego obenemos una lista vacía. Simplemente convierte una lista vacía con el tipo `[a]` a una lista vacía con el tipo `[b]`.

Los tipos que pueden actuar como una caja pueden ser funtores. Puede pensar en una lista como una caja que tiene un número ilimitado de pequeños compartimientos y puden estar todos vacíos, o pueden estár algunos llenos. Asi que, ¿Qué más tiene la propiedad de comportarse como una caja? Por ejemplo, el tipo `Maybe a`. De algún modo, es como una caja que puede o bien no contener nada, en cuyo caso su valor será `Nothing`, o puede contener algo, como `"HAHA"`, en cuyo caso su valor ser`á `Just "HAHA"`. Aquí tienes como `Maybe` es un funtor:

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

De nuevo, fíjate que hemos escrito `instance Functor Maybe where` en lugar de `instance Functor (Maybe m) where`, como hicimos cuando utilizamos la clase `YesNo` junto con `Maybe`. `Functor` quiere un constructor de tipos que tome un tipo y no un tipo concreto. Si mentalemente remplazas las`f` con `Maybe`, `fmap` actua como `(a -> b) -> Maybe a -> Maybe b`para este tipo en particular, lo cual se ve bien. Pero si remplazas `f` con`(Maybe m)`, entonces parecerá que actua como `(a -> b) -> Maybe m a ->Maybe m b`, lo cual no tiene ningún maldito sentido ya que `Maybe` toma un solo parámetro.

De cualquier forma, la implementación de `fmap` es muy simple. Si es un valor vacío o `Nothing`, entonces simplemente devolvemos `Nothing`. Si mapeamos una caja vacía obtenemos una caja vacía. Tiene sentido. De la misma forma que si mapeamos una lista vacía obtenemos un lista vacía. Si no es un valor vacío, sino más bien un único valor envuelto por `Just`, entonces aplicamos la función al contenido de `Just`.

```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Otra cosa que puede ser mapeada y por tanto puede tener una instancia de`Functor` es nuestro tipo `Tree a`. También puede ser visto como una caja (contiene varios o ningún valor) y el constructor de tipos `Tree` toma exactamente un parámetro de tipo. Si vemos la función `fmap` como si fuera una función hecha exclusivamente para `Tree`, su declaración de tipo sería como `(a -> b) -> Tree a -> Tree b`. Vamos a utilizar la recursión con éste. Mapear un árbol vacío poducirá un árbol vacío. Mapear un árbol no vacío producirá un árbol en el que la función será aplicada al elemento raíz y sus sub-árboles derechos e izquierdos serán los mismos sub-árboles, solo que serán mapeado con la función.

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x leftsub rightsub) = Node (f x) (fmap f leftsub) (fmap f rightsub)
```

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (Node 12 EmptyTree (Node 20 EmptyTree EmptyTr
```

¡Bien! ¿Qué pasa con `Either a b`? ¿Puede ser un funtor? La clase de tipos`Functor` quiere constructores de tipos que tomen un solo parámetro de tipo pero `Either` toma dos. Mmm... ¡Ya se! aplicaremos parcialmente `Either`suministrando un solo parámetro de forma que solo tenga un parámetro libre. Aquí tienes como el tipo `Either a` es un funtor en las librerías estándar.

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

Bueno, bueno ¿Qué hemos hecho aquí? Pudes ver como hemos creado una instancia para `Either a` en lugar de para solo `Either`. Esto es así porque`Either a` es un constructor de tipos que toma un parámetro, mientras que`Either` toma dos. Si `fmap` fuese específicamente para `Either a`entonces su declaración de tipo sería `(b -> c) -> Either a b -> Either a c`ya que es lo mismo que `b -> c) -> (Either a) b -> (Either a) c`. En la implementación, mapeamos en el caso del constructor de tipos `Right`, pero no lo hacemos para el caso de `Left`. ¿Por qué? Bueno, si volvemos atrás para ver como se define el tipo `Either a b`, varíamos algo como:

```
data Either a b = Left a | Right b
```

Bueno, si quisieramos mapear una función sobre ambos, `a` y `b` deberían tener el mimso tipo. Quiero decir, si quisieramos mapear una función que toma una cadena y devuelve otra cadena y `b` fuese una cadena pero `a`fuese un número, ésto no funcionaria. También, viendo `fmap` si operara solo con valores de `Either`, veríamos que el primer parámetro tiene que permanecer igual mientras que el segundo puede variar y el primer parámetro está asociado al constructor de datos `Left`.

Esto también encaja con nuestra analogía de las cajas si pensamos en `Left`como una especie de caja vacía con un mensaje de error escrito en un lado diciendonos porque la caja está vacía.

Los diccionarios de `Data.Map` también son funtores ya que pueden contener (o no) valores. En el caso de `Map k v`, `fmap` mapearía una función`v -> v'` sobre un diccionario `Map k v` y devolvería un diccionario con el tipo `Map k v'`.

Nota
Fíjate que `'` no tiene ningún significado especial en los tipos de la misma forma que no tienen ningún significado especial a la hora de nombrar valores. Se suele utilizar para referirse a cosas que son similares, solo que un poco cambiadas.

¡Trata de imaginarte como se crea la instancia de `Map k` para `Functor` tú mismo!

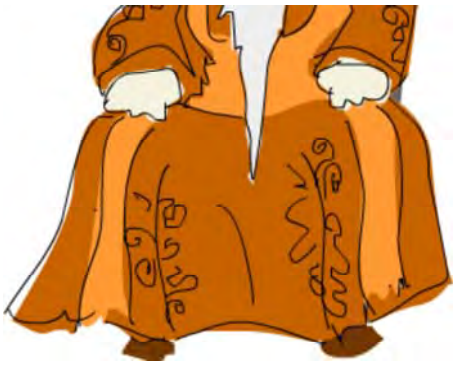
Con la clase de tipos `Functor` hemos visto como las clases de tipos puden representar conceptos de orden superior interesantes. También hemos tenido un poco de práctica aplicando parcialmente tipos y creando instancias. En uno de los siguientes capítulos veremos algunas de las leyes que se aplican a los funtores.

Nota
Los funtores deben obedecer algunas leyes de forma que tengan unas propiedades de las que podamos depender para no tener que pensar mucho luego. Si usamos `fmap (+1)` sobre un la lista `[1,2,3,4]` esperemamos obtener `[2,3,4,5]` y no su inversa, `[5,4,3,2]`. Si usamos `fmap (\a -> a)` (la función identidad, que simplemente devuelve su parámetro) sobre un lista, esperamos obtener la misma lista como resultado. Por ejemplo, si le damos una instancia erronea a nuestro tipo `Tree`, al usar `fmap` en un árbol donde el sub-árbol izquierdo de un nodo solo contenga elementos menores que el nodo y el sub-árbol derecho solo contenga elementos mayores que el nodo podría producir un árbol donde no se cumpliera esto. Veremos la leyes de los funtores con más detalle en un próximo capítulo.

Familias y artes marciales

Los constructores de tipos toman otros tipos como parámetros y terminan produciendo tipos concretos. Esto me recuerda a las funciones, las cuales toman valores como parámetros y producen valores. Hemos visto que los constructores de tipos pueden ser parcialmente aplicados (`Either String` es un constructor de tipos que toma un tipo y devuelve un tipo concreto, como`Either String Int`), al igual que la funciones. Muy interesante. En esta sección, definiremos formalmente como los tipos son aplicados a los

constructores de tipos, de la misma definiremos formalmente como los valores son aplicados a las funciones utilizando declaraciones de tipo. **No necesitas leer esta sección para continuar con tu búsqueda de la sabiduría sobre Haskell** y no consigues entenderlo, no te preocupes. Sin embargo, si lo haces conseguiras un conocimiento profundo del sistema de tipos.



Así que, valores como `3`, `"YEAH"` o `takeWhile` (las funciones también son valores ya que podemos usarlas como parámetros) tienen sus correspondientes tipos. Los tipos son una pequeña etiqueta que llevan los valores de forma que nos permitan razonar sobre estos. Pero los tipos tienen sus propias pequeñas etiquetas, llamadas **familias**. Una familia es más o menos el tipo de un tipo. Puede sonar un poco enrevesado y confuso, pero en realidad es un concepto muy intersante.

¿Qué son las familias y para que son útiles? Bueno, vamos a examinar la familia de un tipo utilizando el comando `:k` en GHCi.

```
ghci> :k Int
Int :: *
```

¿Una estrella? Intrigante... ¿Qué significa? Una `*` significa que el tipo es un tipo concreto. Un tipo concreto es un tipo que no toma ningún parámetro de tipo y valores solo pueden tener tipos que sean tipos concretos. Si tuviera que leer `*` en voz alta (hasta ahora no he tenido que hacerlo), diría*estrella* o simplemente *tipo*.

Vale, ahora vamos a ver cual es la familia de `Maybe`.

```
ghci> :k Maybe
Maybe :: * -> *
```

El constructor de tipos `Maybe` toma un tipo concreto (como `Int`) y luego devuelve un tipo concreto como `Maybe Int`. Y esto es lo que la familia nos está diciendo. De la misma forma que `Int -> Int` representa una función que toma un `Int` y devuelve un `Int`, `* -> *` representa un constructor de tipos que toma un tipo concreto y devuelve otro tipo concreto. Vamos a aplicar el parámetro de tipo a `Maybe` y ver cual es su familia.

```
ghci> :k Maybe Int
Maybe Int :: *
```

¡Justo como esperaba! Hemo pasado un parámetro de tipo a `Maybe` y hemos obtenido un tipo concreto (esto es lo que significa `* -> *`). Un símil (aunque no equivalente, los tipos y las familias son dos cosas distintas) sería si hicieramos `:t isUpper` y `:t isUpper 'A'`. `isUpper` tiene el tipo `Char -> Bool` y `isUpper 'A'` tiene el tipo `Bool` ya que su valor es básicamente `True`.

Utilizamos `:k` con un tipo para obtener su familia, de la misma forma que utilizamos `:t` con un valor para obtener su tipo. Como ya hemos dicho, los tipos son las etiquetas de los valores y las familias son las etiquetas de los tipos y hay similitudes entre ambos.

Vamos a ver otra familia.

```
ghci> :k Either
Either :: * -> * -> *
```

¡Aha! Esto nos dice que `Either` toma dos tipos concretos como parámetros de tipo y produce un tipo concreto. También se parece a una declaracion de tipo de una función que toma dos valores y devuelve algo. Los construcotores de tipos están currificados (como las funciones), así que podemos aplicarlos parcialmente.

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

Cuando quisimos que `Either` formara parte de la clase de tipos `Functor`, tuvimos que aplicarlo parcialmente ya que `Functor` quiere tipos que tomen un solo parámetro`,` mientras que `Either` toma dos. En otras palabras,`Functor` quiere tipos de la familia `* -> *` y por eso tuvimos que aplicar parcialmente `Either` para obtener una familia `* -> *` en lugar de su familia original `* -> * -> *`. Si vemos la definición de `Functor` otra vez

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

veremos que la variable de tipo `f` es utiliza como un tipo que que toma un tipo y produce un tipo concreto. Sabemos que produce un tipo concreto porque es utilizada como el tipo de un valor en una función. Podemos deducir que los tipos que quieren amigables con `Functor` debe ser de la familia `* -> *`.

Ahora vamos a practicar un poco de artes marciales. Echa un vistazo a la clase de tipos que voy a utilizar:

```
class Tofu t where
    tofu :: j a -> t a j
```

Parece complicado ¿Cómo podríamos crear un tipo que tuviera una instancia para esta clase de tipos estraña? Bueno, vamos a ver que familia tiene que tener. Como `j a` es utilizado como el tipo del valor que la función `tofu` toma como parámetro, `j a` debe tener la familia `*`. Asumimos `*` para `a`de forma que podemos inferir que `j` pertenece a la familia `* -> *`.Vemos que `t` también tiene que producir un tipo concreto y toma dos tipos. Sabiendo que `a` es de la familia `*` y `j` de `* -> *`, podemos inferir que `t` es de la familia `* -> (* -> *) -> *`. Así que toma un tipo concreto (`a`), un constructor de tipos (`j`) que toma un tipo concreto y devuelve un tipo concreto. Wau.

Vale, vamos a crear un tipo con una familia `* -> (* -> *) -> *`. Aquí tienes una posible solución.

```
data Frank a b  = Frank {frankField :: b a} deriving (Show)
```

¿Cómo sabemos que este tipo pertenece a la familia `* -> (* -> *) -> *`?Bueno, los campos de un TDA (tipos de datos algebraicos, *ADT* en inglés) sirven para contener valores, así que obviamente pertenecen a la familia`*`. Asumimos `*` para `a`, lo que significa que `b` toma un parámetro de tipo y por lo tanto pertenece a la familia `* -> *`. Ahora que sabemos las familia de `a` y `b` ya que son parámetros de `Frank`, vemos que`Frank` pertenece a la familia `* -> (* -> *) -> *`. El primer `*`representa `a` y `(* -> *)` representa `b`. Vamos a crear algunos valores de `Frank` y comprobar sus tipos.

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YES"}
Frank {frankField = "YES"} :: Frank Char []
```

Mmm... Como `frankField` tiene el tipo en forma de `a b`, sus valores deben tener tipos de forma similar. Puede ser como `Just "HAHA"`, el cual tiene el tipo `Maybe [Char]` o puede ser como `['Y','E','S']` que tiene el tipo `[Char]` (si usaramos

nuestro tipo de listas que creamos anteriormente, sería `List Char`). Y vemos que los tipos de los valores de`Frank` se corresponden con la familia de `Frank`. `[Char]` pertenece a la familia `*` y `Maybe` pertenece a `* -> *`. Como para poder tener valores un tipo tiene que ser un tipo concreto y por lo tanto debe ser completamente aplicado, cada valor de `Frank bla blaaa` pertenece a la familia `*`.

Crear la instancia de `Frank` para `Tofu` es bastante simple. Hemos visto que `tofu` toma un `j a` (que por ejemplo podría ser `Maybe Int`) y devuelve un `t j a`. Así que si remplazamos `j` por `Frank`, el tipo del resultado sería `Frank Int Maybe`.

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

No es muy útil, pero hemos calentado. Vamos a continuar haciedo artes marciales. Tenemos este tipo:

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

Y ahora queremos crear una instancia para la clase `Functor`. `Functor` requiere tipos cuya familia sea `* -> *` pero `Barry` no parece que pertenezca a esa familia. ¿Cúal es la familia de `Barry`? Bueno, vemos que toma tres parámetros de tipo, así que va ser algo como `algo -> algo -> algo-> *`. Esta claro que `p` es un tipo concreto y por lo tanto pertenece a la familia `*`. Para `k` asumimos `*` y por extensión, `t` pertenece a`* -> *`. Ahora solo tenemos que remplazar estas familia por los *algos* que hemos utilizado y veremos que el tipo pertenece a la familia `(* -> *) -> *``-> * -> *`. Vamos a comprobarlo con GHCi.

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ah, teníamos razón. Ahora, para hacer que este tipo forme parte de la clase`Functor` tenemos que aplicar parcialmente los dos primeros parámetros de tipo de forma que nos quedemos con `* -> *`. Esto significa que comenzaremos con nuestra declaración de instancia así: `instance Functor (Barry a b)where`. Si vemos `fmap` como si estuviese hecho exclusivamente para`Barry`, tendría un tipo `fmap :: (a -> b) -> Barry c d a -> Barry c d b`, ya que simplemente hemos remplazado la `f` de `Functor` por `Barry c d`. El tercer parámetro de tipo de `Barry` tendría que cambiar y de esta forma tendríamos:

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

¡Ahí lo tienes! Simplemente hemos aplicado `f` sobre el primer campo.

En esta sección, hemos dado un buen vistazo a como funcionan los parámetros de tipos y como se formalizan con la familias, de la misma forma que formalizamos los parámetros de las funciones con las declaraciones de tipo. Hemos visto que hay similitudes entre las funciones y los constructores de tipos. De todas formas, son cosas totalmente distintas. Cuando trabajamos con Haskell, normalmente no debes preocuparte por la familias ni inferir mentalmente las familias como hemos hecho aquí. Lo normal es que tengas que aplicar parcialmente tu tipo a `* -> *` o `*` cuando creamos una instancia para alguna clase de la librería estándar, pero está bien saber como funciona realmente. Es interesante saber que los tipos tienen sus propios pequeños tipos también. De nuevo, no tienes porque entender todo lo que acabamos de hacer aquí, pero si entiendes como funcionan las familias, tienes más posibilidades de entender correctamente el sistema de tipos de Haskell.

Entrada y salida

Ya hemos mencionado que Haskell es un lenguaje de programación puramente funcional. Mientras que en un lenguaje imperativo normalmente consigues resultados dando al computador una serie de pasos ha ejecutar, la programación funcional es más como definir que son las cosas. En Haskell, una función no puede cambiar un estado, como cambiar el contenido de una variable (cuando una función cambia de estado, decimos que tiene *efectos secundarios*). La única cosa que una función de Haskell puede hacer es devolvernos algún resultado basándose en los parámetros que le demos. Si una función es llamada dos veces con los mismo parámetros, tiene que devolver el mismo resultado. Mientras esto puede parecer un poco restrictivo desde el punto de vista de un mundo imperativo, ya hemos visto como en realidad es algo genial. En un lenguaje imperativo no tienes ninguna garantía de que una función que solo debe jugar con unos números no vaya a quemar tu casa, secuestrar a tu perro o rallar tu coche con una patata mientras juega con esos números. Por ejemplo, cuando hacemos una búsqueda binaria con un árbol, no insertamos ningún elemento en el árbol modificando algún nodo. Nuestra función para insertar un elemento en un árbol en realidad devuelve un nuevo árbol, ya que no puede modificar el árbol anterior.

Como el hecho de que las funciones no sean capaces de cambiar el estado es algo bueno, ya que nos ayuda a razonar acerca de nuestros programas, existe un problema con esto. Si una función no puede cambiar nada en el mundo ¿Cómo se supone que nos va a decir el resultado que ha calculado? Para conseguir que nos diga lo que ha calculado, tiene que cambiar el estado de un dispositivo de salida (normalmente el estado de la pantalla), lo cual emitirá fotones que viajaran por nuestro cerebro para cambiar el estado de nuestra mente, impresionante.

No te desesperes, no está todo perdido. Haskell en realidad tiene un sistema muy inteligente para tratar con funciones que tienen efectos secundarios de forma que separa la parte de nuestro programa que es pura de la parte de nuestro programa que es impura, la cual hace todo el trabajo sucio de hablar con el teclado y la pantalla. Con estas partes bien separadas, podemos seguir razonando acerca de nuestro programa puro y tomar ventaja de todo lo que nos ofrece la pureza, como la evaluación perezosa, seguridad y modularidad mientras nos comunicamos con el mundo exterior.

¡Hola mundo!

Hasta ahora, siempre hemos cargado nuestras funciones en GHCi para probarlas y jugar con ellas. También hemos explorado las funciones de la librería estándar de esta forma. Pero ahora, después de ocho capítulos, por fin vamos a escribir nuestro primer programa de Haskell real ¡Wau! Y por supuesto, vamos a crear el mítico `"¡hola, mundo!"`.

Nota

A efectos de este capítulo, voy a asumir que estás utilizando un sistema *unix* para aprender Haskell. Si estás en *Windows*, te sugiero que bajes cygwin, el cual es un entorno *Linux* para *Windows*, o dicho de otra forma, justo lo que necesitas.

Así que para empezar, pega lo siguiente en tu editor de texto favorito:

```
main = putStrLn "hello, world"
```

Acabamos de definir la función `main` y dentro de ella llamamos a una función `putStrLn` con el parámetro `"hello, world"`. Parece algo corriente, pero no lo es, como veremos dentro de un momento. Guarda el fichero como `helloworld.hs`.

Y ahora vamos a hacer algo que no hemos hecho nunca antes ¡Vamos a compilar un programa! ¿No estás nervioso? Abre un terminal y navega hasta el directorio donde se encuentra `helloworld.hs` y haz lo siguiente:

```
$ ghc --make helloworld
[1 of 1] Compiling Main             ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

¡Vale! Con un poco de suerte habrás obtenido algo similar y podrás ejecutar el programa haciendo `./helloworld`.

```
$ ./helloworld
hello, world
```

Ahí lo tienes, nuestro primer programa compilado que muestra un mensaje por la terminal ¡Extraordinariamente aburrido!

Vamos a examinar lo que hemos escrito. Primero, vamos a ver el tipo de la función `putStrLn`.

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

Podemos leer el tipo de `putStrLn` como: `putStrLn` toma una cadena y devuelve una acción `IO` que devuelve un tipo `()` (es decir, la tupla vacía, también conocida como unidad). Una acción `IO` es algo que cuando se realiza, cargará con una acción con algún efecto secundario (como leer desde la entrada o mostrar cosas por pantalla) y contendrá algún tipo de de valor dentro de él. Mostrar algo por pantalla realmente no tiene ningún tipo de valor resultado, así que se utiliza el valor ficticio `()`.

Nota
La tupla vacía tiene el valor de `()` y también tiene el tipo`()`. Algo como `data Nada = Nada`.

Y ¿Cuándo se ejecuta una acción `IO`? Bueno, aquí es donde entra en juego`main`. Una acción `IO` se ejecuta cuando le damos el nombre` `main` y ejecutamos nuestro programa.

Tener todo tu programa en una sola acción `IO` puede parecer un poco restringido. Por este motivo podemos usar la sintaxis `do` para juntar varias acciones `IO` en una. Echa un vistazo al siguiente ejemplo:

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

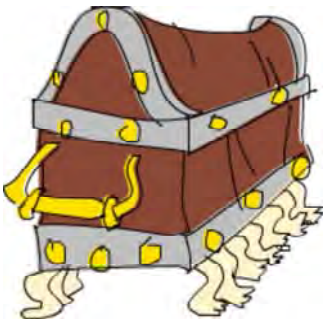
Ah... Interesante ¡Sintaxis nueva! Se lee de forma similar a la de un programa imperativo. Si lo compilas y lo ejecutas, probablemente se comportará como esperas. Fíjate que hemos utilizado un `do` y luego hemos puesto una serie de pasos, exactamente igual que en un programa imperativo. Cada uno de estos pasos es una acción `IO`. Poniéndolas todas ellas juntas

en un mismo bloque`do` conseguimos una sola acción `IO`. La acción que obtenemos tiene el tipo `IO ()` porque ese es el tipo de la última acción dentro del bloque.

Por este motivo, `main` siempre tiene que tener el tipo `IO algo`, donde`algo` es algún tipo concreto. Por convenio, no se suele especificar la declaración de tipo de `main`.

Una cosa interesante que no hemos visto antes está en la tercera línea, la cual es `name <- getLine`. Parece como si leyera una línea de texto y la guardara en una variable llamada `name` ¿De verdad? Bueno, vamos a examinar el tipo de `getLine`.

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



Vale. `getLine` es una acción `IO` que contiene un resultado del tipo`String`. Parece que tiene sentido ya que esperará a que el usuario escriba algo en la terminal y luego ese algo será representado con una cadena. Entonces ¿Qué pasa con `name <- getLine`? Puedes leer ese trozo de código como: realiza la acción `getLine` y luego liga el resultado al valor`name`. `getLine` tiene el tipo `IO String`, así que `name` tendrá el tipo `String`. Puedes imaginar una acción `IO` como una caja con patas que saldrá al mundo real y hará algo allí (como pintar un grafiti en una pared) y quizá vuelva con algún dato dentro. Una vez se ha traído ese dato, la única forma de abrir la caja y tomar el dato de su interior es utilizando la construcción `<-`. Y si estamos extrayendo datos de una acción `IO`, solo podemos sacarlos cuando estemos dentro de alguna acción `IO`. Así es como Haskell gestiona y separa las partes puras e impuras de nuestro código. En ese sentido `getLine` es impuro ya que el resultado no está garantizado que sea el mismo cuando se llamada dos veces. Este es el porqué su resultado está *contaminado* con constructor de tipos `IO` y solo podemos extraer estos datos dentro de un código `IO`. Y como el código `IO` está contaminado también, cada cálculo que dependa en un dato contaminado con`IO` tendrá un resultado contaminado también.

Cuando decimos *contaminado*, no lo decimos en el sentido de que nunca más podremos usar el resultado contenido en una acción `IO` en nuestro código puro. No, cuando ligamos un valor contenido en una acción `IO` a un nombre lo *descontaminamos* temporalmente. Cuando hacemos `nombre <- getLine`,`nombre` es una cadena normal, ya que representa lo que hay dentro de la caja. Podemos tener un función realmente complicada que, digamos, toma tu nombre (una cade normal) como parámetro y predice tu suerte y todo tu futuro basándose únicamente en tu nombre. Podría ser algo así:

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

`tellFortune` (o cualquier otra función a la que se le pase `name`) no tiene porque saber nada acerca de `IO`, es simplemente una función normal de tipo `String -> String`.

Mira este trozo de código ¿Es válido?

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

Si has dicho que no, puedes ir a por una galletita. Si haz dicho que sí, ves olvidándote de caprichos. La razón por la que esto no funciona es que `++` requiere que sus dos parámetros sean del mismo tipo lista. El parámetro de la izquierda tiene el tipo `String` (o `[Char]` si lo prefieres), mientras que `getLine` tiene el tipo `IO String`. No podemos concatenar una cadena con una acción `IO`. Primero debemos extraer el resultado de la acción`IO` para obtener un valor del tipo `String` y la única forma de conseguirlo es haciendo algo como `name <- getLine` dentro de una acción`IO`. Si queremos tratar con datos impuros tenemos

que hacerlo en un entorno impuro. La mancha de la impureza se propaga al igual que una plaga por nuestro código y es nuestro deber mantener las partes `IO` tan pequeñas como sean posibles.

Cada acción `IO` que es ejecutada tiene un resultado encapsulado con él. Por este motivo podríamos haber escrito el código anterior como:

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

Sin embargo, `foo` simplemente tendría el valor `()` lo cual no es de mucha utilidad. Fíjate que no hemos ligado el último `putStrLn` a ningún nombre. Esto es debido a que en un bloque `do`, **la última acción no puede ser ligada** como las dos primeras. Cuando nos aventuremos en el mundo de las mónadas veremos el motivo concreto de esta restricción. Por ahora, puedes pensar que un bloque `do` extrae automáticamente el valor de la última acción y lo liga a su propio resultado.

Excepto para la última línea, cada línea de un bloque `do` que no se liga puede también escribirse como una ligadura. Así que `putStrLn "Blah"` se puede escribir como `_ <- putStrLn "Blah"`. Sin embargo es algo inútil, por lo que no usamos `<-` para acciones que no contienen un resultado importante, como `putStrLn algo`.

Los principiantes a veces piensan que hacer cosas como `name = getLine`leerá una línea por la entrada y la ligará a `name`. Bueno, pues no, lo que hace esto es darle a la acción `getLine` un nuevo nombre, llamado `name`. Recuerda que para obtener el valor contenido dentro de una acción `IO`, tienes que ligarlo a un nombre con `<-` dentro de otra acción `IO`.

Las acciones `IO` solo son ejecutadas cuando se les dan el nombre de`main` o cuando están dentro de una acción `IO` más grande que hemos compuesto con un bloque `do`. Podemos usar un bloque `do` para juntar algunas acciones `IO` y luego usar esa acción `IO` dentro de otro bloque `do` y así sucesivamente. De cualquier modo, al final solo se ejecutarán cuando sean alcanzadas por `main`.

Oh, cierto, también hay otro caso en el que las acciones `IO` son ejecutadas. Cuando escribimos una acción `IO` en GHCi y pulsamos intro.

```
ghci> putStrLn "HEEY"
HEEY
```

Incluso cuando escribimos un número o una llamada a una función en GHCi, éste lo evaluará (tanto como necesite) y luego llamará a `show` para mostrar esa cadena en la terminal utilizando `putStrLn` de forma implícita.

¿Recuerdas las secciones `let`? Si no, refresca tu memoria leyendo esta*sección*. Tienen la forma `let ligaduras in expresión`, donde`ligaduras` son los nombres que se les dan a las expresiones y `expresión` será la expresión donde serán evaluadas. También dijimos que las listas por comprensión no era necesaria la parte `in`. Bueno, puedes usarlas en un bloque `do` prácticamente igual que en las listas por comprensión. Mira esto:

```
mport Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " " ++ bigLastName ++ ", how are you?"
```

¿Ves como las acciones `IO` dentro del bloque `do` están alineadas? Fíjate también en como la sección `let` está alineada con las acciones `IO` y los nombres de `let` están alineados entre ellos. Es una buena práctica hacer esto, ya que el sangrando es importante en Haskell. Hemos hecho `map toUpperfirstName`, lo cual convierte algo como `"john"` en la cadena `"JOHN"`. Hemos ligado esa cadena en mayúsculas a un nombre y luego la hemos utilizado en una cadena para mostrarla por la terminal.

Puedes estar preguntándote cuando utilizar `<-` y cuando utilizar `let`. Bueno, recuerda que `<-` es (por ahora) para ejecutar acciones `IO` y ligar sus resultados. Sin embargo, `map toUpper firstName` no es una acción`IO`. Es una expresión pura de Haskell. Así que utilizamos `<-` cuando queremos ligar los resultados de una acción `IO` mientras que usamos `let`para ligar expresiones puras. Si hubiéramos hecho algo como `let firstName =getLine`, simplemente hubiéramos dado un nuevo nombre a la acción `getLine`y seguiríamos necesitado utilizar `<-` para ejecutar la acción.

Ahora vamos a crear un programa que lee continuamente una línea y muestra esa línea con sus palabras al revés. La ejecución del programa se detendrá cuando encuentre una línea vacía. Aquí tienes el programa.

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

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

Para entender como funciona, puedes ejecutar el programa antes de leer el código.

Nota

Para ejecutar un programa puedes o bien compilarlo produciendo un ejecutable y luego ejecutarlo usando `ghc --make helloworld` y luego `./helloworld` o bien puedes usar el comando `runhaskell`así: `runhaskell helloworld.hs` y tu programa será ejecutado al vuelo.

Primero vamos a echar un vistazo a la función `reverseWords`. Es solo una función normal que toma una cadena como `"hey there man"` y luego llama a `words` lo cual produce una lista de palabras como`["hey","there","man"]`. Luego mapeamos `reverse` sobre la lista, obteniendo `["yeh","ereht","nam"]`, luego volvemos a tener una sola cadena utilizando `unwords` y el resultado final es `"yeh ereht nam"`. Fíjate en como hemos utilizado la composición de funciones. Sin la composición de funciones tendríamos que haber escrito algo como `reverseWords st = unwords(map reverse (words st))`.

¿Qué pasa con `main`? Primero, obtenemos una línea del terminal ejecutando`getLine` y la llamamos `line`. Y ahora tenemos una expresión condicional. Recuerda que en Haskell, cada `if` debe tener su `else` ya que toda expresión debe tener algún tipo de valor. Usamos la condición de forma que cuando sea cierta (en nuestro caso, para cuando la línea esté vacía) realicemos una acción `IO` y cuando no lo es, realizamos la acción ubicada en el `else`. Por este motivo las condiciones dentro de una acción `IO` tienen la forma `if condición then acción else acción`.

Vamos a echar un vistazo a lo que pasa bajo la cláusula `else`. Como debemos tener exactamente una sola acción `IO` después del `else` tenemos que usar un bloque `do` para juntar todas la acciones en una. También podía ser escrito así:

```
else (do
    putStrLn $ reverseWords line
    main)
```

Esto hace más explícito el hecho de que un bloque `do` sea visto como una sola acción `IO`, pero es más feo. De cualquier modo, dentro del bloque`do` llamamos a `reverseWords` sobre la línea que obtuvimos de `getLine`y luego mostramos el resultado por la terminal. Luego de esto, simplemente ejecutamos `main`. Es llamado de forma recursiva y no hay ningún problema ya que `main` es por si mismo una acción `IO`. De cierto modo es como si volviéramos al inicio del programa.

Ahora ¿Qué sucede cuando `null line` se evalúa a cierto? Se ejecuta la acción que está después del `then`. Si buscamos veremos que pone `then`return ()`. Si conoces algún lenguaje imperativo como *C*, *JavaPython*, probablemente estés pensando que ya sabes lo que es `return` y que puedes saltarte este párrafo tan largo. Bueno, pues **el `return` de Haskell no tiene nada que ver con el `return` de la mayoría de los otros lenguajes**. Tiene el mismo nombre, lo cual confunde a mucha a gente, pero en realidad es muy diferente. En los lenguajes imperativos, `return`normalmente termina la ejecución de un método o una subrutina y devuelve algún tipo de valor a quien quiera que lo llamó. En Haskell (dentro de la acciones`IO` concretamente), lo que hace es convertir un valor puro en una acción`IO`. Si lo piensas como en la analogía de la caja que vimos, `return` toma un valor y lo pone dentro de una caja. La acción `IO` resultante realmente no hace nada, simplemente tiene dicho valor como resultado. Así que en un contexto `IO`, `return "haha"` tendrá el tipo `IO String` ¿Cuál es el motivo de transformar un valor puro en una acción que realmente no hace nada? ¿Por qué contaminar más nuestro programa con `IO`? Bueno, necesitamos alguna acción `IO` en caso de que encontremos una línea vacía. Por este motivo hemos creado una acción `IO` que realmente no hace nada con `return()`.

Al utilizar `return` no causamos que un bloque `do` termine su ejecución ni nada parecido. Por ejemplo, este programa ejecutará hasta la última línea sin ningún problema.

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

Todo lo que estos `return` hacen es crear acciones `IO` que en realidad no hacen nada excepto contener un valor, el cual es desperdiciado ya que no se liga a ningún nombre. Podemos utilizar `return` en combinación con `<-`para ligar cosas a nombres.

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

Como puedes ver, `return` es en cierto modo lo opuesto de `<-`. Mientras que `return` toma valores y los mete en una caja, `<-` toma una caja (y la ejecuta) y saca el valor que contiene, enlazándolo a un nombre. Sin embargo hacer estas cosas es un poco redundante, ya que puedes utilizar secciones`let` para conseguir lo mismo:

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

Cuando tratemos con bloques `do` `IO`, normalmente utilizamos `return` o bien porque queremos crear una acción `IO` que no haga nada o bien porque queremos que el resultado que albergue la acción `IO` resultante de un bloque `do` no sea el valor de la última acción.

Nota

Un bloque `do` puede contener una sola acción `IO`. En ese caso, es lo mismo que escribir solo dicha acción. Hay gente que prefiere escribir `then do return ()` en este caso ya que el `else` también tiene un `do`.

Antes de que veamos como tratar con ficheros, vamos a echar un vistazo a algunas funciones que son útiles a la hora de trabajar con `IO`.

- `putStr` es muy parecido a `putStrLn` en el sentido de que toma una cadena y devuelve una acción que imprimirá esa cadena por la terminal, solo que `putStr` no salta a una nueva línea después de imprimir la cadena tal y como `putStrLn` hace.

```
main = do   putStr "Hey, "
            putStr "I'm "
            putStrLn "Andy!"
```

```
$ runhaskell putstr_test.hs
Hey, I'm Andy!
```

Su tipo es `putStr :: String -> IO ()`, así que el resultado contenido en la acción `IO` es la unidad. Un valor inútil, por lo que no tiene sentido ligarlo a nada.

- `putChar` toma un carácter y devuelve una acción `IO` que lo imprimirá por la terminal.

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

```
$ runhaskell putchar_test.hs
teh
```

`putStr` en realidad está definido recursivamente con ayuda de`putChar`. El caso base es la cadena vacía, así que si estamos imprimiendo la cadena vacía, simplemente devolvemos una acción `IO` que no haga nada utilizando `return ()`. Si no esta vacía, imprimimos el primer carácter de la cadena utilizando `putChar` y luego imprimimos el resto de la cadena usando `putStr`.

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

Fíjate en que podemos utilizar la recursión en `IO` de la misma forma que lo hacemos en el código puro. Al igual que en el código puro, definimos el caso base y luego pensamos que es realmente el resultado. Es una acción que primero imprime el primer carácter y luego imprime el resto de la cadena.

- `print` toma un valor de cualquier tipo que sea miembro de la clase`Show` (por lo que sabemos que se puede representar como una cadena), llama a `show` con ese valor para obtener su representación y luego muestra esa cadena por la terminal. Básicamente es `putStrLn . show`. Primero ejecuta `show` con un valor y luego alimenta `putStrLn` con ese valor, lo cual devuelve una acción que imprimirá nuestro valor.

```
main = do   print True
            print 2
            print "haha"
            print 3.2
            print [3,4,3]
```

```
$ runhaskell print_test.hs
True
```

```
2
"haha"
3.2
[3,4,3]
```

Como puedes ver, es una función muy útil ¿Recuerdas cuando hablamos de que las acciones `IO` se ejecutan solo cuando son alcanzadas por `main` o cuando intentamos evaluarlas en GHCi? Cuando escribimos un valor (como `3` o `[1,2,3]`) y pulsamos intro, GHCi en realidad utiliza `print`con ese valor para mostrarlo por la terminal.

```
ghci> 3
3
ghci> print 3
3
ghci> map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
ghci> print (map (++"!") ["hey","ho","woo"])
["hey!","ho!","woo!"]
```

Cuando queremos imprimir cadenas normalmente utilizamos `putStrLn` ya que solemos querer las dobles comillas que rodean la representación de una cadena, pero para mostrar valores de cualquier otro tipo se suele utilizar`print`.

- `getChar` es una acción `IO` que lee un carácter por la entrada estándar (teclado). Por ello, su tipo es `getChar :: IO Char`, ya que el resultado contenido dentro de la acción `IO` es un carácter. Ten en cuenta que debido al *buffering*, la acción de leer un carácter no se ejecuta hasta que el usuario pulsa la tecla intro.

```
main = do
    c <- getChar
    if c /= ' '
        then do
            putChar c
            main
        else return ()
```

Este programa parece que debe leer un carácter y comprobar si es un espacio. Si lo es, detiene la ejecución del programa y si no lo es, lo imprime por la terminal y luego repite su ejecución. Bueno, parece que hace esto, pero no lo hace de la forma que esperamos. Compruébalo.

```
$ runhaskell getchar_test.hs
hello sir
hello
```

La segunda línea es la salida. Hemos escrito `hello sir` y luego hemos pulsado intro. Debido al *buffering*, la ejecución del programa solo empieza después de ejecutar intro y no después de cada carácter pulsado. Una vez pulsamos intro, actúa como si hubiéramos escrito esos caracteres desde el principio. Intenta jugar un poco con este programa para entender como funciona.

- La función `when` se encuentra en el módulo `Control.Monad` (para acceder a ella haz `import Control.Monad`). Es interesante ya que dentro de un bloque `do` parece como si fuese una sentencia de control de flujo, pero en realidad es una función normal. Toma un valor booleano y una acción`IO` de forma que si el valor booleano es `True`, devolverá la misma acción que le suministremos. Sin embargo, si es falso, nos devolverá una acción `return ()`, acción que no hace absolutamente nada. Aquí tienes como podríamos haber escrito el trozo de código anterior que mostraba el uso de `getChar` utilizando `when`:

```
import Control.Monad

main = do
    c <- getChar
```

```
    when (c /= ' ') $ do
        putChar c
        main
```

Como puedes ver, es útil para encapsular el patrón *if algo then do acción else return ()*`. También existe la función `unless` que es exactamete igual a `when` solo que devuleve la acción original cuando ecuentra`False` en lugar de `True`.

- `sequence` toma una lista de acciones `IO` y devuevle una acción que realizará todas esas acciones una detrás de otra. El resultado contenido en la acción `IO` será una lista con todos los resultados de todas las acciones `IO` que fueron ejecutadas. Su tipo es `sequence :: [IO a] ->`IO [a]`. Hacer esto:

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

Es exactamente lo mismo que hacer:

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

Así que `sequence [getLine, getLine, getLine]` crea una acción `IO` que ejecutará `getLine` tres veces. Si ligamos esa acción a un nombre, el resultado será una lista que contendrá todos los resultados, en nuestro caso, una lista con tres líneas que haya introducido el usuario.

Un uso común de `sequence` es cuando mapeamos funciones como `print` o`putStrLn` sobre listas. Al hacer `map print [1,2,3,4]` no creamos una acción `IO`. Creará una lista de acciones `IO`, ya que es lo mismo que si escribiéramos `[print 1, print 2, print 3, print 4]`. Si queremos transformar esa lista de acciones en una única acción IO, tenemos que secuenciarla.

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

¿Qué es eso de `[(),(),(),(),()]`? Bueno, cuando evaluamos una acción`IO` en GHCi es ejecutada y su resultado se muestra por pantalla, a no ser que el resultado sea `()`, en cuyo caso no se muestra. Por este motivo al evaluar `putStrLn "hehe"` GHCi solo imprime `"hehe"` (ya que el resultado contenido en la acción `putStrLn "hehe"` es `()`). Sin embargo cuando utilizamos `getLine` en GHCi, el resultado de esa acción si es impreso por pantalla, ya que `getLine` tiene el tipo `IO String`.

- Como mapear una función que devuelve una acción `IO` sobre una lista y luego secuenciarla es algo muy común, se introdujeron las funciones auxiliares `mapM` y `mapM_`. `mapM` toma una función y una lista, mapea la función sobre la lista y luego la secuencia. `mapM_` hace lo mismo, solo que después se deshace del resultado. Normalmente utilizamos`mapM_` cuando no nos importa el resultado de las acciones secuenciadas.

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

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

- `forever` toma una acción `IO` y devuelve otra acción `IO` que simplemente repetirá la primera acción indefinidamente. Está situada en`Control.Monad`. Este pequeño programa preguntará al usuario por una cadena y luego la devolverá en mayúsculas, indefinidamente:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

- `forM` (situado en `Control.Monad`) es como `mapM` solo que tiene sus parámetros cambiados de sitio. El primer parámetro es la lista y el segundo la función a mapear sobre la lista, la cual luego será secuenciada ¿Para qué es útil? Bueno, con un uso creativo de funciones lambda y la notación `do` podemos hacer cosas como estas:

```
import Control.Monad

main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Which color do you associate with the number " ++ show a ++
        color <- getLine
        return color)
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

`(\a -> do ... )` es una función que toma un número y devuelve una acción`IO`. Tenemos que rodearla con paréntesis, ya que de otro modo la función lambda pensaría que las dos últimas líneas le pertenecen. Fíjate que usamos`return color` dentro del bloque `do`. Lo hacemos así para que la acción `IO` que define el bloque `do` tenga como resultado el color que deseamos. Realmente no tenemos que hacerlos porque `getLine` ya lo tienen contenido. Hacer `color <- getLine` para luego hacer `returncolor` es simplemente extraer el resultado de `getLine` para luego volver a insertarlo otra vez, así que es lo mismo que hacer solo`getLine`.``forM`` (llamado con sus dos parámetros) produce una acción`IO`, cuyo resultado ligaremos a `colors`. `colors` es una simple lista que contiene cadenas. Al final, imprimimos todos esos colores haciendo `mapM putStrLn colors`.

Puedes verlo en el sentido de que `forM` crea una acción `IO` para cada elemento de una lista. Lo que haga cada acción dependerá del elemento que haya sido utilizado para crear la acción. Al final, realiza todas esas acciones y liga todos los resultados a algo. No tenemos porque ligarlo, podemos simplemente desecharlo.

```
$ runhaskell form_test.hs
Which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

En realidad podríamos haber hecho lo mismo sin `forM`, solo que con`forM` es más legible. Normalmente usamos `forM` cuando queremos mapear y secuenciar algunas acciones que hemos definido utilizando la notación `do`. Del mismo modo, podríamos haber remplazado la última línea por `forM colors putStrLn`.

En esta sección hemos aprendido las bases de la entrada y salida. También hemos visto que son las acciones `IO`, como nos permiten realizar acciones de entrada y salida y cuando son realmente ejecutadas. Las acciones `IO` son valores al igual que cualquier otro valor en Haskell. Podemos pasarlas como parámetros en las funciones y las funciones pueden devolver acciones como resultados. Lo que tiene de especial es cuando son alcanzadas por `main` (o son el resultado de una sentencia en GHCi), son ejecutadas. Y es en ese momento cuando pueden escribir cosas en tu pantalla o reproducirYakety Saxpor tus altavoces. Cada acción `IO` también puede contener un resultado que nos dirá que ha podido obtener del mundo real.

No pienses en la función `putStrLn` como una función que toma una cadena y la imprime por pantalla. Piensa que es una función que toma una cadena y devuelve una ación `IO`. Esa acción `IO`, cuando sea ejecutada, imprimirá por pantalla dicha cadena.

Ficheros y flujos de datos

`getChar` es una acción de E/S que lee un solo carácter desde la terminal.`getLine` es una acción de E/S que lee una línea desde la terminal. Estas funciones son bastante sencillas y la mayoría de lenguajes tienen funciones o sentencias similares. Pero ahora vamos a ver `getContents`.`getContents` es una acción de E/S que lee cualquier cosa de la entrada estándar hasta que encuentre un carácter de fin de fichero. Su tipo es`getContents :: IO String`. Lo bueno de `getContents` es que realiza una E/S perezosa. Cuando hacemos `foo <- getContents`, no lee todos los datos de entrada de golpe, los almacena en memoria y luego los liga a `foo`. No ¡Es perezoso! Dirá “Sí, sí, ya leeré la entrada de la terminal luego, cuando de verdad lo necesites”.

`getContents` es realmente útil cuando estamos redirigiendo la salida de un programa a la entrada de otro programa. En caso de que no sepas como funciona la redirección en sistemas *unix*, aquí tienes una pequeña introducción. Vamos a crear un fichero de texto que contenga este pequeñohaiku:

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

Sí, tienes razón, este haiku apesta. Si conoces alguna buena guía sobre haikus házmelo saber.

Ahora recuerda aquel pequeño programa que escribimos cuando explicamos la función `forever`. Le pedía al usuario una línea y la devolvía en mayúsculas, luego volvía a hace lo mismo indefinidamente. Solo para que no tengas que desplazarte hacia atrás, aquí tienes el código de nuevo:

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
```

```
    l <- getLine
    putStrLn $ map toUpper l
```

Vamos a guardar este programa como `capslocker.hs` o algo parecido y lo compilamos. Y ahora, vamos a utilizar redirecciones *unix* para suministrar nuestro fichero de texto directamente a nuestro pequeño programa. Nos vamos a ayudar del uso del programa GNU `cat`, el cual muestra por la terminal el contenido del fichero que le pasemos como parámetro ¡Mira!

```
$ ghc --make capslocker
[1 of 1] Compiling Main             ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ cat haiku.txt
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

Como puedes ver, para redireccionar la salida de un programa (en nuestro caso`cat`) a la entrada de otro (`capslocker`) se consigue con el carácter`|`. Lo que acabamos de hacer sería equivalente a ejecutar `capslocker`, escribir nuestro haiku en la terminal y luego introducir el carácter de fin de fichero (normalmente esto se consigue pulsando *Ctrl+D*). Es como ejecutar`cat haiku.txt` y decir: “Alto espera, no muestres esto por pantalla, pásaselo a `capslocker`”.

Así que lo que estamos haciendo al utilizar `forever` es básicamente tomar la entrada y transformarla en algún tipo de salida. Por este motivo podemos utilizar `getContents` para hacer nuestro programa mejor e incluso más corto.

```
import Data.Char

main = do
    contents <- getContents
    putStr (map toUpper contents)
```

Ejecutamos la acción de E/S `getContents` y nombramos la cadena que produce como `contents`. Luego, trazamos `toUpper` sobre la cadena y mostramos el resultado por la terminal. Ten en cuenta que las cadenas son básicamente listas, las cuales son perezosas, y `getContents` es una acción de E/S perezosa. Por lo tanto no intentará leer todo el contenido de golpe para guardarlo en memoria antes de mostrarlo en mayúsculas por la terminal. En realidad mostrará la versión en mayúsculas conforme vaya leyendo, ya que solo lee una línea de la entrada cuando realmente lo necesita.

```
$ cat haiku.txt | ./capslocker
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

Genial, funciona ¿Qué pasaría si ejecutamos `capslocker` e intentamos escribir líneas de texto nosotros mismos?

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

Salimos pulsando *Ctrl+D*. Como ves, muestra nuestra entrada en mayúsculas línea por línea. Cuando el resultado de `getContents` se liga a`contents`, no se representa en memoria como una cadena real, si no más bien como una promesa de que al final producirá una cadena. Cuando trazamos`toUpper` sobre `contents`, también es una promesa de que se trazará esa función sobre el contenido final. Por último, cuando se ejecuta `putStr` le dice a la promesa anterior: “Hey ¡Necesito una línea en

mayúsculas!". Entonces es cuando en realidad `getContents` lee la entrada y le pasa una línea al código que le ha pedido que produzca algo tangible. Ese código traza`toUpper` sobre esa línea y le pasa el resultado a `putStr`, y ésta se encarga de mostrarla. Luego `putStr` dice: "Hey, necesito la siguiente línea ¡Vamos!" y se repite hasta que no hay mas datos en la entrada, lo cual se representa con el carácter de fin de fichero.

Vamos a crear un programa que tome algunas líneas y luego solo muestre aquellas que tengan una longitud menor de 10 caracteres. Observa:

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

shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Hemos hecho la parte de nuestro programa dedicada a E/S tan pequeña como a sido posible. Ya que nuestro programa se supone que toma una entrada y muestra una salida basándose en la entrada, podemos implementarlo leyendo los contenidos de la entrada, ejecutando una función sobre ellos y luego mostramos lo que nos devuelve esa función.

La función `shortLinesOnly` funciona así: toma una cadena, como`"short\nlooooooooooooooong\nshort again"`. Esta cadena tiene tres líneas, dos de ellas son cortas y la del medio es larga. Ejecuta la función `lines` sobre esa cadena, de forma que obtenemos `["short", "looooooooooooooong",``"short again"]` que luego ligamos a `allLines`. Luego esta lista de cadenas es filtrada de forma que solo las líneas que sean menores de 10 caracteres de longitud permanecen en la lista, produciendo `["short","short again"]`. Finalmente `unlines` concatena la lista en una única cadena, devolviendo `"short\nshort again"`. Vamos a probarlo.

```
i'm short
so am i
i am a looooooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
looooooooooooooooooooooooooong
short
```

```
$ ghc --make shortlinesonly
[1 of 1] Compiling Main             ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
$ cat shortlines.txt | ./shortlinesonly
i'm short
so am i
short
```

Redireccionamos los contenidos de `shortlines.txt` a la entrada de`shortlinesonly`, de forma que obtenemos únicamente las líneas cortas.

Este patrón de tomar una cadena como entrada, transformarla con una función y mostrar el resultado de esa transformación es tan común que existe una función que hace esto más fácil, la función `interact`. `interact` toma una función del tipo `String -> String` como parámetro y devuelve una acción de E/S que tomara la entrada del programa, ejecutará la función sobre ella y mostrará por pantalla el resultado de esta función. Vamos a modificar nuestro programa para que utilice esta función.

```
main = interact shortLinesOnly
```

```
shortLinesOnly :: String -> String
shortLinesOnly input =
    let allLines = lines input
        shortLines = filter (\line -> length line < 10) allLines
        result = unlines shortLines
    in  result
```

Con el propósito de mostrar que podemos conseguir lo mismo con mucho menos código (incluso aunque sea un poco menos legible) y demostrar nuestras habilidades de composición de funciones, vamos a modificarlo un poco más.

```
main = interact $ unlines . filter ((<10) . length) . lines
```

Wau ¡Lo hemos reducido a una única línea de código!

`interact` se puede utilizar para crear programas a los que se les redireccionará algún contenido y luego mostrará un resultado, o para crear programas que parezcan que leen una línea escrita por el usuario desde la entrada, muestren un resultado basándose en esa línea y luego continúen con otra línea. En realidad no hay ninguna diferencia entre ellos, simplemente depende de como lo use el usuario.

Vamos a crear un programa que lea continuamente una línea y nos diga si esa línea es un palíndromo o no. Podríamos simplemente utilizar `getLine` para leer una línea, mostrar al usuario si es palíndroma o no, y volver a ejecutar`main`. Pero es más simple si utilizamos `interact`. Cuando utilices `interact`, piensa en que tienes que hacer para transformar la entrada del programa en la salida que deseas. En nuestro caso, tenemos que remplazar cada línea de la entrada en `"palindrome"` o `"not a palindrome"`. Así que tenemos que transformar algo como `"elephant\nABCBA\nwhatever"` en`"not a palindrome\npalindrome\nnot a palindrome"` ¡Vamos a intentarlo!

```
respondPalindromes contents = unlines (map f (lines contents))
    where isPalindrome xs = xs == reverse xs
          f xs = if isPalindrome xs then "palindrome" else "not a palindrome"
```

Vamos a escribirlo en estilo libre de puntos:

```
respondPalindromes = unlines . map f . lines
    where isPalindrome xs = xs == reverse xs
          f xs = if isPalindrome xs then "palindrome" else "not a palindrome"
```

Sencillo. Primero convierte algo como `"elephant\nABCBA\nwhatever"` en`["elephant", "ABCBA", "whatever"]` y luego traza `f` sobre la lista, devolviendo `["not a palindrome", "palindrome", "not a palindrome"]`. Por último utiliza `unlines` para concatenar la lista de cadenas en una sola cadena. Ahora podemos hacer:

```
main = interact respondPalindromes
```

Vamos a comprobarlo.

```
$ runhaskell palindromes.hs
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

Incluso aunque hemos creado un programa que transforma una gran cadena de entrada en otra, actúa como si hubiéramos hecho un programa que lee línea a línea. Esto se debe a que Haskell es perezoso y quiere mostrar la primera línea del

resultado, pero no lo puede hacer porque aún no tiene la primera línea de la entrada. Así que tan pronto tenga la primera línea de la entrada, mostrará la primera línea de la salida. Salimos del programa utilizando el carácter de fin de fichero.

También podemos utilizar el programa redireccionando el contenido de un fichero. Digamos que tenemos este fichero:

```
dogaroo
radar
rotor
madam
```

Y lo hemos guardado como `words.txt`. Así sería como redireccionaríamos el fichero a la entrada de nuestro programa.

```
$ cat words.txt | runhaskell palindromes.hs
not a palindrome
palindrome
palindrome
palindrome
```

De nuevo, obtenemos la misma salida que si hubiésemos ejecutado nuestro programa y hubiésemos tecleado nosotros mismos las palabras. Simplemente no vemos la entrada de `palindromes.hs` porque ha sido redireccionada desde un fichero.

Probablemente ya sepas como funciona E/S perezosa y como se puede aprovechar. Puedes pensar en términos que como se supone que debe ser la salida y escribir una función que haga la transformación. En la E/S perezosa, no se consume nada de la entrada hasta que realmente tenga que hacerse, es decir, cuando queramos mostrar por pantalla algo que depende de la entrada.

Hasta ahora, hemos trabajado con la E/S mostrando y leyendo cosas de la terminal ¿Pero qué hay de escribir y leer ficheros? Bueno, de cierto modo, ya lo hemos hecho. Se puede pensar que leer algo desde la terminal es como leer algo desde un fichero especial. Lo mismo ocurre a la hora de escribir en la terminal, es parecido a escribir en un fichero. Podemos llamar a estos dos ficheros `stdout` y `stdin`, que representan la salida estándar y la entrada estándar respectivamente. Teniendo esto en cuenta, veremos que escribir y leer ficheros es muy parecido a escribir en la salida estándar y leer desde la entrada estándar.

Empezaremos con un programa realmente simple que abre un fichero llamado`girlfriend.txt`, que contiene un verso del éxito Nº 1 de *Avril Lavigne*,*Girlfriend*, y lo muestra por la terminal. Aquí tienes `girlfriend.txt`:

```
Hey! Hey! You! You!
I don't like your girlfriend!
No way! No way!
I think you need a new one!
```

Y aquí tienes nuestro programa:

```
import System.IO

main = do
    handle <- openFile "girlfriend.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Ejecutándolo, obtenemos el resultado esperado:

```
$ runhaskell girlfriend.hs
Hey! Hey! You! You!
I don't like your girlfriend!
```

```
No way! No way!
I think you need a new one!
```

Vamos a analizarlo línea a línea. La primera línea son solo cuatro exclamaciones que intentan llamar nuestra atención. En la segunda línea, Avril nos dice que no le gusta nuestra actual pareja. La tercera línea tiene como objetivo enfatizar su desacuerdo, mientras que la cuarta nos sugiere que busquemos una nueva novia.

¡Genial! Ahora vamos a analizar nuestro programa línea a línea. El programa tiene varias acciones de E/S unidas en un bloque `do`. En la primera línea del bloque `do` vemos que hay una función nueva llamada `openFile`. Su tipo es el siguiente: `openFile :: FilePath -> IOMode -> IO Handle`. Si lo lees en voz alta dice: `openFile` toma la ruta de un fichero y un `IOMode`y devuelve una acción de E/S que abrirá el fichero indicado y contendrá un manipulador como resultado.

`FilePath` es simplemente un *sinónimo de tipo* de`String`, se define como:

```
type FilePath = String
```

`IOMode` es un tipo que se define como:

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

De la misma forma que aquel tipo que creamos que representaba los siete días de la semana, este tipo es una enumeración que representa lo que queremos hacer con un fichero abierto. Muy simple. Fíjate que el tipo es `IOMode` y no `IO Mode`. `IO Mode` sería una acción de E/S que contendría un valor del tipo `Mode` como resultado, pero `IOMode` es simplemente una enumeración.

Al final esta función devuelve una acción de E/S que abrirá el fichero indicado del modo indicado. Si ligamos la acción a algo al final obtenemos un `Handle`. Un valor del tipo `Handle` representa donde está nuestro fichero. Lo usaremos para manipular el fichero de forma que sepamos de donde leer y escribir datos. Sería un poco estúpido abrir un fichero y no ligar el manipulador ya que no podríamos hacer nada con ese fichero. En nuestro caso ligamos el manipulador a `handle`.

En la siguiente línea vemos una función llamada `hGetContents`. Toma un`Handle`, de forma que sabe de donde tiene que leer el contenido, y devuelve una `IO String`, una acción de E/S que contiene como resultado el contenido del fichero. Esta función se parece mucho a `getContents`. La única diferencia es que `getContents` lee automáticamente desde la entrada estándar (es decir desde la terminal), mientras que `hGetContents` toma el manipulador de un fichero que le dice donde debe leer. Por lo demás, funcionan exactamente igual. Al igual que `getContents`, `hGetContents` no leerá todo el contenido de un fichero de golpe si con forme lo vaya necesitando. Esto es muy interesante ya que podemos tratar a `contents` como si fuera todo el contenido del fichero, solo que en realidad no estará cargado en la memoria. En caso de que leyéramos un fichero enorme, ejecutar`hGetContents` no saturaría la memoria ya que solo se leerá lo que se vaya necesitando.

Fíjate en la diferencia entre el manipulador utilizado para representar el fichero y los contenidos del fichero, ligados en nuestro programa a `handle`y `contents`. El manipulador es algo que representa el fichero con el que estamos trabajando. Si te imaginas el sistema de ficheros como si fuera un gran libro y cada fichero fuera un capítulo del libro, el manipulador sería como un marcador que nos indica por donde estamos leyendo (o escribiendo) en un capítulo, mientras que el contenido sería el capítulo en si.

Con `putStr contents` simplemente mostramos el contenido del fichero por la salida estándar. Luego ejecutamos `hClose`, el cual toma un manipulador y devuelve una acción de E/S que cierra el fichero ¡Tienes que cerrar tu mismo cada fichero que abras con `openFile`!

Otra forma de hacer lo que mismo que acabamos de hacer es utilizando la función `withFile`, cuya declaración de tipo es `withFile :: FilePath ->IOMode -> (Handle -> IO a) -> IO a`. Toma la ruta de un fichero, un`IOMode` y luego toma una función que a su vez toma un manipulador y devuelve una acción de E/S. `withFile` devuelve una acción de E/S que abrirá el fichero indicado, hará algo con él y luego cerrará el fichero. El resultado contenido en la acción de E/S final es el mismo que el resultado contenido en la acción de E/S de la función que se le pasa como parámetro. Quizá te suente un poco complicado, pero es realmente simple, especialmente con la ayuda de las lambdas. Aquí tienes nuestro programa anterior reescrito utilizando `withFile`:

```
import System.IO

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

Como puedes ver ambos son muy parecidos. `(\handle -> ... )` es la función que toma un manipulador y devuleve una acción de E/S y de forma habitual esta función se implementea utilizando lambdas. La razón por la que debe tomar una función que devuelva una acción de E/S en lugar de tomar directamente una acción de E/S para hacer algo y luego cerrar el fichero, es para que la función que le pasemos sepa sobre que fichero operar. De este modo,`withFile` abre un fichero y le pasa el manipulador a la función que le demos. Obtiene una acción de E/S como resultado y luego crear una acción de E/S que se comporte de la misma forma, solo que primero cierra el fichero. Así sería como implementaríamos la función `withFile`:

```
withFile' :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

Sabemos que el resultado debe ser una acción de E/S así que podemos empezar directamente con un `do`. Primero abrimos el fichero y obtenemos el manipulador. Luego aplicamos `handle` a nuestra función y obtenemos una acción de E/S que realizará todo el trabajo. Ligamos esa acción a `result`, cerramos el fichero y hacemos `return result`. Al realizar el `return` sobre el resultado que contenia la acción de E/S que obtuvimos de `f`, hacemos que nuestra acción de E/S contenga el mismo resultado que obtuvimos de `f handle`. Así que si `f handle` devuleve una acción que lea un número de líneas de la entrada estándar y luego las escriba en el fichero, de forma que contenga como resultado el número de líneas que ha leído, la acción resultante de `withFile'` también tendrá como resultado el número de líneas leidas.

De la misma forma que `hGetContents` funciona igual que `getContents` pero sobre el fichero indicado, existen también `hGetLine`, `hPutStr`,`hPutStrLn`, `hGetChar`, etc. Funcionan exactamente igual que sus homónimas, solo que toman un manipulador como parámetro y operar sobre el fichero indicado en lugar de sobre la entrada o salida estándar. Por ejemplo, `putStrLn` es una función que toma una cadena y devuelve una acción de E/S que mostrará esa cadena por la terminal seguida de un salto de línea.`hPutStrLn` toma un manipulador y una cadena y devuelve una acción de E/S que escribirá esa cadena en el fichero indicado, seguido de un salto de línea. Del mismo modo, `hGetLine` toma un manipulador y devuelve una acción de E/S que lee una línea de su fichero.

Cargar ficheros y luego tratar sus conteidos como cadenas es algo tan común que tenemos estas tres pequeñas funciones que hacen nuestra vida más fácil:

- `readFile` tiene la declaración de tipo `readFile :: FilePath -> IOString`. Recueda, `FilePath` es solo un sinónimo de `String`.`readFile` toma la ruta de un fichero y devuelve un acción de E/S que leerá ese fichero (de forma perezosa) y ligará sus contenidos a una cadena. Normalmente es más cómodo que hacer `openFile` y ligar su manipulador para luego utilizar `hGetContents`. Aquí tienes como sería nuestro ejemplo anterior utilizando `readFile`:

```
import System.IO

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

  Como no obtenemos un manipulador con el cual identificar nuestro fichero, no podemos cerrarlo manualmente, así que Haskell se encarga de cerrarlo por nosotros cuando utilizamos `readFile`.

- `writeFile` tiene el tipo `FilePath -> String -> IO ()`. Toma la ruta de un fichero y una cadena que escribir en ese fichero y devuelve una acción de E/S que se encargará de escribirla. En caso de que el fichero indicado ya exista, sobreescribirá el fichero desde el incio. Aquí tienes como convertir `girlfriend.txt` en una versión en mayúsculas y guardarlo en `girlfriendcaps.txt`:

```
import System.IO
import Data.Char

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

```
$ runhaskell girlfriendtocaps.hs
$ cat girlfriendcaps.txt
HEY! HEY! YOU! YOU!
I DON'T LIKE YOUR GIRLFRIEND!
NO WAY! NO WAY!
I THINK YOU NEED A NEW ONE!
```

- `appendFile` tiene el mismo tipo que `writeFile`, solo que `appendFile` no sobreescribe el fichero desde el principio en caso de que el fichero indicado ya exista, sino que añade contiendo al final del fichero.

  Digamos que tenemos un fichero `todo.txt` que contiene una tarea que debemos realizar en cada línea. Ahora vamos a crear un programa que tome una línea por la entrada estándar y la añada a nuestra lista de tareas.

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

```
$ runhaskell appendtodo.hs
Iron the dishes
$ runhaskell appendtodo.hs
Dust the dog
$ runhaskell appendtodo.hs
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

Tenemos que añadir `"\n"` al final de cada línea ya que `getLine` no nos devuelve el carácter de fin de línea al final.

Oh, una cosa más. Hemos hablado de como al hacer `contents <- hGetContentshandle` no se provoca que el fichero enetero sea leído de golpe y guardado en memoria. Es una acción de E/S perezosa, así que al hacer esto:

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

En realidad es como redireccionar el fichero a la salida. De la misma forma que puedes tratar las cadenas como flujos de datos también puedes tratar los ficheros como flujos de datos. Esto leerá una línea cada vez y la mostrará por pantalla. Probablemente te estes preguntado ¿Con qué frecuencia se accede al disco? ¿Qué tamaño tiene cada transferencia? Bueno, para ficheros de texto, el tamaño por defecto para el búfer es una línea. Esto siginfica que la parte más pequeña que se puede leer de fichero de una sola vez es una línea. Por este motivo el ejemplo anterior en realidad leía una línea, la mostraba, leía otra línea, la mostraba, etc. Para ficheros binarios, el tamaño del búfer suele ser de un bloque. Esto significa que los ficheros binarios se leen de bloques en bloques. El tamaño de un bloque es el que le apetezca a tu sistema operativo.

Puedes controlar como se comporta exactamente el búfer utilizando la función`hSetBuffering`. Ésta toma un manipulador y un `BufferMode` y devuelve una acción de E/S que estable las propiedades del búfer para ese fichero.`BufferMode` es una simple tipo de enumeración y sus posibles valores son:`NoBuffering`, `LineBuffering` or `BlockBuffering (Maybe Int)`. El`Maybe Int` indica el tamaño del bloque, en bytes. Si es `Nothing`, el sistema operativo determinará el tamaño apropiado. `NoBuffering` significa que se escribirá o se leera un carácter cada vez. Normalmente `NoBuffering` no es muy eficiente ya que tiene que acceder al disco muchas veces.

Aquí tienes nuestro ejemplo anterior, solo que esta vez leerá bloques de 2048 bytes en lugar de línea por línea.

```
main = do
    withFile "something.txt" ReadMode (\handle -> do
        hSetBuffering handle $ BlockBuffering (Just 2048)
        contents <- hGetContents handle
        putStr contents)
```

Leer ficheros con bloques grandes nos puede ayudar si queremos minimizar el acceso a disco o cuando nuestro fichero en realidad es un rescurso de una red muy lenta.

También podemos utilizr `hFlush`, que es una función que toma un manipulador y devuelve una acción de E/S que vaciará el búfer del fichero asociado al manipulador. Cuando usamos un búfer de líneas, el búfer se vacía depués de cada línea. Cuando utilizmos un búfer de bloques, el búfer se vacía depués de que se lea o escriba un bloque. También se vacía después de cerrar un manipulador. Esto signfica que cuando alcanzemos un salto de línea, el mecanismode de lectura (o escritura) informará de todos los datos hasta el momento. Pero podemos utilizar `hFlush` para forzar ese informe de datos. Depués de realizar el vaciado, los datos están disponibles para cualquier otro programa que este ejecutandose.

Para entender mejor el búfer de bloques, imagina que la taza de tu retrete está configurada para vaciarse cuando alcance los cuatro litros de agua en su interior. Así que empiezas a verter agua en su interior y cuando alcanza la marca de los cuatro litros automaticamente se vacía, y los datos que contenian el agua que has vertido hasta el momento son leidos o escritos. Pero también puedes vaciar manualmente el retrete pulsando el botón que éste posee. Esto hace que el retrete se vacie y el agua (datos) dentro del retrete es leida o escrita. Solo por si no te has dado cuenta, vacia manualmente el retrete es una metáfora para `hFlush`. Quizá este no sea una buena analogía en el mundo de las analogías estándar de la programación, pero quería un objeto real que se pudiera vaciar.

Ya hemos creado un programa que añade una tarea a nuestra lista de tareas pendientes `todo.txt`, así que ahora vamos a crear uno que elimine una tarea. Voy a mostrar el código a continuación y luego recorerremos el progeama juntos para que veas que es realmente fácil. Usaremos una cuantas funciones nuevas que se encuentran en `System.Directory` y una funcón nueva de`System.IO`.

De todas formas, aquí tienes el programa que elimina una tarea de`todo.txt`:

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

main = do
    handle <- openFile "todo.txt" ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    putStr $ unlines numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

Primero abrirmos el fichero `todo.txt` en modo lectura y ligamos el manipulador a `handle`.

A continuación, utilizamos una función que aún no conocemos y que proviene de `System.IO`, `openTempFile`. Su nombre es bastante auto descriptivo. Toma la ruta de un directorio temporal y una plantilla para nombres para un fichero y abre un fichero temporal. Hemos utilizado `"."` para el directorio temporal porque `"."` representa el directorio actual en cualquier S.O. Utilizamos `"temp"` como plantilla para el nombre del fichero, de forma que que el fichero temporal tendra como nombre *temp* más algunos caracteres aleatorios. Devuelve una acción de E/S que crea un fichero temporal y el resultado de esa acción es una dupla que contiene: el nombre del fichero temporal y el manipulador asociado a ese fichero. Podríamo haber abierto algún fichero normal como `todo2.txt` o algo parecido pero es un práctica mejor utilizar `openTempFile` y asegurarse así de que no sobreescribimos nada.

La razón por la que no hemos utilizado `getCurrentDirectory` para obtener el directorio actual y luego pasarselo a `openTempFile` es porque `"."` representa el directorio actual tanto es sitemas *unix* como en *Windows*.

Luego ligamos los contenido de `todo.txt` a `contents`. Después dividimos esa cadena en una lista de cadenas, una cadena por línea. Así que`todoTasks` es ahora algo como `["Iron the dishes", "Dust the dog",``"Take salad out of the oven"]`. Unimos los números desde el 0 en adelante y esa lista con una función que toma un número, digamos 3, y una cadena, como `"hey"`, así que `numberedTasks` sería `["0 - Iron the dishes", "1 -`Dust the dog" ...`. Concatenamos esa lista de cadenas en una sola cadena delimitada por saltos de línea con `unlines` y la mostramos por la terminal. Fíjate que en lugar de hacer esto podríamos haber hecho algo como `mapMputStrLn numberedTasks`.

Le preguntamos al usuario que tarea quiere eliminar y esperamos que introduzca un número. Digamos que queremos eliminar la número 1, que es `Dust the dog`, así que introducimos `1`. `numberString` es ahora `"1"` y como queremos un número y no una cadena, utilizamos `read` sobre ella para obtener un`1` y ligarlo a `number`.

Intenta recordar las funciones `delete` y `!!` del módulo `Data.List`.`!!` devuelve un elemento de una lista dado un índice y `delete` elimina la primera ocurrencia de un elemento en una lista, y devuelve una nueva lista sin dicho elemento.

`(todoTasks !! number)`, con `number` a `1`, devuelve `"Dust the dog"`. Ligamos `todoTasks` sin la primera ocurrencia de `"Dust the dog"` a `newTodoItems` y luego unimos todo en una sola cadena utilizando `unlines` antes de escribirlo al fichero temporal que hemos abierto. El fichero original permanece sin cambios y el fichero temporal ahora contiene todas las tareas que contiene el original, excepto la que queremos eliminar.

Después de cerrar ambos ficheros, tanto el original como el temporal, eliminamos el original con `removeFile`, que, como puedes ver, toma la ruta de un fichero y lo elimina. Después de eliminar el `todo.txt` original, utilizamos `renameFile` para renombrar el fichero temporal a`todo.txt`. Ten cuidad, tanto `removeFile` como `renameFile` (ambas contenidas en `System.Directory`) toman rutas de ficheros y no manipuladores.

¡Y eso es todo! Podríamos haberlo hecho en menos líneas, pero tenemos cuidado de no sobreescribir ningún fichero existente y preguntamos educadamente al sistema operativo que nos diga donde podemos ubicar nuestro fichero temporal ¡Vamos a probarlo!

```
$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
1

$ cat todo.txt
Iron the dishes
Take salad out of the oven

$ runhaskell deletetodo.hs
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0

$ cat todo.txt
Take salad out of the oven
```

Parámetros de la línea de comandos

Prácticamente es una obligación trabajar con parámetros de la línea de comandos cuando estamos creando un programa que se ejecuta en la terminal. Por suerte, la biblioteca estándar de Haskell tiene una buena forma de obtener los parámetros de la línea de comandos.

En la sección anterior, creamos un programa para añadir tareas a nuestra lista de tareas pendientes y otro programa para eliminar tareas de dicha lista. Hay dos problemas con el enfoque que tomamos. La primera es que fijamos el nombre del fichero que contenía la lista en nuestro código fuente. Simplemente decidimos que sería `todo.txt` y el usuario nunca podría trabajar con varias listas.

Una forma de solventar este problema sería preguntar siempre al usuario con que lista trabajar. Utilizamos este enfoque cuando quisimos saber que tarea quería el usuario eliminar. Funciona, pero hay mejores opciones ya que requiere que el usuario ejecute el programa, espere a que el programa le pregunte algo y luego decirle lo que necesita. A esto se llama

programa interactivo y el problema de los programas interactivos es: ¿Qué pasa si queremos automatizar la ejecución del programa? Como en un fichero de comandos por lotes que ejecuta un programa o varios de ellos.

Por este motivo a veces es mejor que el usuario diga al programa que tiene que hacer cuando lo ejecuta, en lugar de que el programa tenga que preguntar al usuario una vez se haya ejecutado. Y que mejor forma de que el usuario diga al programa que quiere que haga cuando se ejecute que con los parámetros de la línea de comandos.

El módulo `System.Environment` tiene dos acciones de E/S muy interesante. Una es `getArgs`, cuya declaración de tipo es `getArgs :: IO [String]`y es una acción de E/S que obtendrá los parámetros con los que el programa fue ejecutado y el resultado que contiene son dichos parámetros en forma de lista. `getProgName` tiene el tipo `IO String` y es una acción de E/S que contiene el nombre del programa.

Aquí tienes un pequeño programa que demuestra el comportamiento de estas funciones:

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn args
   putStrLn "The program name is:"
   putStrLn progName
```

Ligamos `getArgs` y `getProgName` a `args` y `progName`. Mostramos`The arguments are:` y luego para cada parámetro en `args` hacemos`putStrLn`. Al final también mostramos el nombre del programa. Vamos a compilar esto como `arg-test`.

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

Bien. Armados con este conocimiento podemos crear aplicaciones de línea de comandos interesantes. De hecho vamos a continuar y a crear una. En la sección anterior creamos programas separados para añadir tareas y para eliminarlas. Ahora vamos a crear un programa con ambas funcionalidades, lo que haga dependerá de los parámetros de la línea de comandos. También vamos a permitir que puede trabajar con ficheros diferentes y no solo `todo.txt`.

Llamaremos al programa `todo` y será capaz de hacer tres cosas:

- Ver las tareas
- Añadir una tarea
- Eliminar una tarea

No nos vamos a preocupar sobre posibles errores en la entrada ahora mismo.

Nuestro programa estará creado de forma que si queremos añadir la tarea`Find the magic sword of power` en el fichero `todo.txt`, tendremos que escribir `todo add todo.txt "Find the magic sword of power"` en la terminal. Para ver las tareas simplemente ejecutamos `todo view todo.txt` y para eliminar la tarea con índice 2 hacemos `todo remove todo.txt 2`.

Empezaremos creando una lista de asociación. Será una simple lista de asociación que tenga como claves los parámetros de la línea de comandos y funciones como sus correspondientes valores. Todas estas funciones serán del tipo `[String] -> IO ()`. Tomarán la lista de parámetros de la línea de comandos y devolverán una acción de E/S que se encarga de mostrar las tareas, añadir una tarea o eliminar una tarea.

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

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

Tenemos que definir `main`, `add`, `view` y `remove`, así que empecemos con `main`.

```
main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args
```

Primero, ligamos los parámetros a `(command:args)`. Si te acuerdas del ajuste de patrones, esto significa que el primer parámetro se ligará con`command` y el resto de ellos con `args`. Si ejecutamos nuestro programa como `todo add todo.txt "Spank the monkey"`, `command` será `"add"` y`args` será `["todo.txt", "Spank the monkey"]`.

En la siguiente línea buscamos el comando en lista de asociación. Como`"add"` se asocia con `add`, obtendremos `Just add` como resultado. Utilizamos de nuevo el ajuste de patrones para extraer esta función del tipo`Maybe` ¿Qué pasaría si el comando no estuviese en la lista de asociación? Bueno, entonces devolvería `Nothing`, pero ya hemos dicho que no nos vamos a preocupar demasiado de los errores en la entrada, así que el ajuste de patrones fallaría y junto a él nuestro programa.

Para terminar, llamamos a la función `action` con el resto de la lista de parámetros. Esto devolverá una acción de E/S que o bien añadirá una tarea, o bien mostrará una lista de tareas, o bien eliminará una tarea. Y como está acción es parte del bloque `do` de `main`, se ejecutará. Si seguimos el ejemplo que hemos utilizado hasta ahora nuestra función `action` será `add`, la cual será llamada con `args` (es decir con `["todo.txt","Spank the monkey"]`) y devolverá una acción que añadirá la tarea `Spankthe monkey` a `todo.txt`.

¡Genial! Todo lo que nos queda ahora es implementar las funciones `add`,`view` y `remove`. Empecemos con `add`:

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

Si ejecutamos nuestro programa como `todo add todo.txt "Spank the monkey"`,`"add"` será ligado a `command` en el primer ajuste de patrones del bloque`main`, mientras que `["todo.txt", "Spank the monkey"]` será pasado a la función que obtengamos de la lista de asociación. Así que como no estamos preocupándonos acerca de posibles entradas erróneas, podemos usar el ajuste de patrones directamente sobre una lista con esos dos elementos y devolver una acción de E/S que añada la tarea al final de fichero, junto con un salto de línea.

A continuación vamos a implementar la funcionalidad de mostrar la lista de tareas. Si queremos ver los elementos de un fichero, ejecutamos `todo viewtodo.txt`. Así que en el primer ajuste de patrones, `command` será`"view"` y `args` será `["todo.txt"]`.

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks
```

Ya hicimos algo muy parecido en el programa que eliminaba tareas a la hora de mostrar las tareas para que el usuario pudiera elegir una, solo que aquí solo mostramos las tareas.

Y para terminar implementamos `remove`. Será muy similar al programa que eliminaba tareas, así que si hay algo que no entiendas revisa la explicación que dimos en su momento. La principal diferencia es que no fijamos el nombre del fichero a `todo.txt` sino que lo obtenemos como parámetro. Tampoco preguntamos al usuario el índice de la tarea a eliminar ya que también lo obtenemos como un parámetro más.

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

Abrimos el fichero basándonos en `fileName` y abrimos un fichero temporal, eliminamos la línea con índice de línea que el usuario desea eliminar, lo escribimos en un fichero temporal, eliminamos el fichero original y renombramos el fichero temporal a `fileName`.

¡Aquí tienes el programa entero en todo su esplendor!

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

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

main = do
    (command:args) <- getArgs
    let (Just action) = lookup command dispatch
    action args

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line) [0..] todoTasks
    putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    handle <- openFile fileName ReadMode
    (tempName, tempHandle) <- openTempFile "." "temp"
    contents <- hGetContents handle
    let number = read numberString
```

```
        todoTasks = lines contents
        newTodoItems = delete (todoTasks !! number) todoTasks
    hPutStr tempHandle $ unlines newTodoItems
    hClose handle
    hClose tempHandle
    removeFile fileName
    renameFile tempName fileName
```

Resumiendo: creamos una lista de asociación que asocie los comandos con funciones que tomen argumentos de la línea de comandos y devuelvan acciones de E/S. Vemos que comando quiere ejecutar el usuario y obtenemos la función apropiada a partir de la lista de asociación. Llamamos a esa función con el resto de parámetros de la línea de comandos y obtenemos una acción de E/S que realizará la acción apropiada cuando sea ejecutada.

En otros lenguajes, deberíamos haber implementado esto utilizando un gran`switch` o cualquier otra cosa, pero gracias a las funciones de orden superior se nos permite crear una lista de asociación que nos devolverá la acción de E/S adecuada para cada comando que pasemos por la línea de comandos.

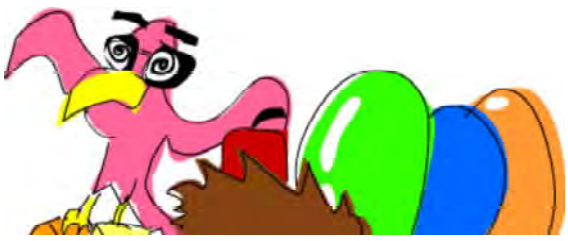
¡Vamos a probar nuestra aplicación!

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven

$ ./todo add todo.txt "Pick up children from drycleaners"

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from drycleaners

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from drycleaners
```

Otra cosa interesante acerca de esto es que bastante sencillo añadir funcionalidad adicional. Simplemente tenemos que agregar un elemento más en la lista de asociación y luego implementar la función correspondiente. Como ejercicio puedes implementar el comando `bump` que tomará un fichero y un y un índice de una tarea y hará que dicha tarea aparezca al principio de la lista de tareas pendientes.

Puedes hacer que este programa fallé de forma más elegante en caso de que reciba unos parámetros erróneos (como por ejemplo `todo UP YOURS HAHAHAH`) creando una acción de E/S que simplemente informe que ha ocurrido un error (digamos `errorExit :: IO ()`) y luego comprar si hay algún parámetro erróneo para realizar el informe. Otra forma sería utilizando excepciones, lo cual veremos dentro de poco.

Aleatoriedad



Muchas veces mientras programamos, necesitamos obtener algunos datos aleatorios. Quizá estemos haciendo un juego en el que se tenga que lanzar un dado o quizá necesitemos generar algunos datos para comprobar nuestro programa. Hay mucho usos para los datos aleatorios. Bueno, en realidad, pseudo-aleatorios, ya que todos sabemos que la única fuente verdadera de aleatoriedad

en un mono sobre un monociclo con un trozo de queso en un mano y su trasero en la otra. En esta sección, vamos a ver como Haskell genera datos aparentemente aleatorios.

En la mayoría de los otros lenguajes, tenemos funciones que nos devuelven números aleatorios. Cada vez que llamas a la función, obtienes (con suerte) un número aleatorio diferente. Bueno, recuerda, Haskell es un leguaje funcional puro. Por lo tanto posee transparencia referencial. Lo que significa que una función, dados los mismo parámetros, debe producir el mismo resultado. Esto es genial ya que nos permite razonar sobre los programas de una forma diferente y nos permite retrasar la evaluación de las operaciones hasta que realmente las necesitemos. Si llamamos a una función, podemos estar seguros de que no hará cualquier otra cosa antes de darnos un resultado. Todo lo que importa es su resultado. Sin embargo, esto hace un poco complicado obtener números aleatorios. Si tenemos una función como:

```
randomNumber :: (Num a) => a
randomNumber = 4
```

No será muy útil como función de números aleatorios ya que siempre nos devolverá el mismo `4`, aunque puedo asegurar que ese 4 es totalmente aleatorio ya que acabo de lanzar un dado para obtenerlo.

¿Qué hacen demás lenguajes para generar número aparentemente aleatorios? Bueno, primero obtienen algunos datos de tu computadora, como la hora actual, cuanto y a donde has movido el ratón, el ruido que haces delante del computador, etc. Y basándose en eso, devuelve un número que parece aleatorio. La combinación de esos factores (la aleatoriedad) probablemente es diferente en cada instante de tiempo, así que obtienes números aleatorios diferentes.

Así que en Haskell, podemos crear un número aleatorio si creamos una función que tome como parámetro esa aleatoriedad y devuelva un número (o cualquier otro tipo de dato) basándose en ella.

Utilizaremos el módulo `System.Random`. Contiene todas las funciones que calmaran nuestra sed de aleatoriedad. Vamos a jugar con una de las funciones que exporta, llamada `random`. Su declaración de tipo es `random ::(RandomGen g, Random a) => g -> (a, g)` ¡Wau! Hay nuevas clases de tipos en esta declaración. La clase de tipos `RandomGen` es para tipos que pueden actuar como fuentes de aleatoriedad. La clase de tipos `Random` es para tipos que pueden tener datos aleatorios. Un dato booleano puede tener valores aleatorios, `True` o `False`. Un número también puede tomar un conjunto de diferentes valores alotarios ¿Puede el tipo función tomar valores aleatorios? No creo. Si traducimos la declaración de tipo de `random` al español temos algo como: toma un generador aleatorio (es decir nuestra fuente de aleatoriedad) y devuelve un valor aleatorio y un nuevo generador aleatorio ¿Por qué devuelve un nuevo generador junto al valor aleatorio? Lo veremos enseguida.

Para utilizar la función `random`, primero tenemos que obtener uno de esos generadores aleatorios. El módulo `System.Random` exporta un tipo interensante llamado `StdGen` que posee una instancia para la clase de tipos `RandomGen`. Podemos crear un `StdGen` manualmente o podemos decirle al sistema que nos de uno basandose en un motón de cosas aleatorias.

Para crear manualmente un generador aletario, utilizamos la función `mkStdGen`. Tiene el tipo `Int -> StdGen`. Toma un entero y basándose en eso, nos devuelve un generador aleatorio. Bien, vamos a intentar utilizar el tandem `random mkStdGen` para obtener un número aleatorio.

```
ghci> random (mkStdGen 100)
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
```

```
    `Random a' arising from a use of `random' at <interactive>:1:0-20
  Probable fix: add a type signature that fixes these type variable(s)
```

¿Qué pasa? Ah, vale, la función `random` puede devolver cualquier tipo que sea miembro de la clase de tipos `Random`, así que tenemos que decir a Haskell exactamente que tipo queremos. Recuerda también que devuelve un valor aleatorio y un generador.

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

¡Por fin, un número que parece aleatorio! El primer componente de la dupla es nuestro número aleatorio mientras que el segundo componente es una representación textual del nuevo generador ¿Qué sucede si volvemos a llamar`random` con el mismo generador?

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

Por supuesto. El mismo resultado para los mismos parámetros. Vamos a probar dándole como parámetro un generador diferente.

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

Genial, un número diferente. Podemos usar la anotación de tipo con muchos otros tipos.

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Vamos a crear una función que simule lanzar una modena tres veces. Si`random` no devolviera un generador nuevo junto con el valor aleatorio, tendríamos que hacer que esta función tomara tres generadores como parámetros y luego devolver un resultado por cada uno de ellos. Pero esto parece que no es muy correcto ya que si un generador puede crear un valor aleatorio del tipo `Int` (el cual puede tener una gran variedad de posibles valores) debería ser capaz de simular tres lazamientos de una moneda (que solo puede tener ocho posibles valores). Así que este es el porqué de que `random`devuelva un nuevo generador junto al valor generado.

Represtaremos el resultado del lanzamiento de una moneda con un simple`Bool`. `True` para cara, `False` para cruz.

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

Llamamos a `random` con el generador que obtivimos como parámetro y obtenemos el resultado de lanzar una moneda junto a un nuevo generador. Luego volvemos a llamar la misma función, solo que esta vez con nuestro nuevo generador, de forma que obtenemos el segundo resultado. Si la hubiéramos llamado con el mismo generador las tres veces, todos los resultados hubieran sido iguales y por tanto solo hubiéramos podido obtener como resultados`(False, False, False)` o `(True, True, True)`.

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

Fíjate que no hemos tendio que hacer `random gen :: (Bool, StdGen)`. Se debe a que ya hemos especificado en la declaración de tipo de la función que queremos valores booleanos. Por este motivo Haskell puede inferir que queremos valores booleanos.

¿Y qué pasaría si quisiéramos lanzar la moneda cuatro veces? ¿Y cinco? Bien, para eso tenemos la función llamada `randoms` que toma un generador y devulve una secuencia infinita de valores aletorios.

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

¿Por qué `randoms` no devuelve un nuevo generador junto con la lista? Podemos implementar la función `randoms` de forma muy sencilla como:

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

Una función recursiva. Obtenemos un valor aleatorio y nuevo generador a parir del generador actual y creamos una lista que tenga el valor aleatorio como cabeza y una lista de valores aloratorios basada en el nuevo generador como cola. Como queremos ser capazes de generar una cantidad infinita valores aleatorios, no podemos devolver un nuevo generador.

Podríamos crear una función que generara secuencias de números aletorios finitas y devolviera también un nuevo generador.

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in  (value:restOfList, finalGen)
```

De nuevo, una funcón recursiva. Decimos que si queremos cero valores alatorios, devolvemos una lista vacía y el generador que se nos dió. Para cualquier otra cantidad de valores aleatorios, primero obtenemos un número aleatorio y nuevo generador. Esto será la cabeza. Luego decimos que la cola será `n-1` valores aleatorios generadors con el nuevo generador. Terminamos devolviendo la cabeza junto el resto de la lista y el generador que obtuvimos cuando generamos los `n-1` valores aleatorios.

¿Y si queremos obtener un valor aleatorio dentro de un determindo rango? Todos los enteros que hemos generador hasta ahora son escandalosamente grandes o pequeños ¿Y si queremos lanzar un dado? Bueno, para eso utilizamos `randomR`. Su declaración de tipo es `randomR :: (RandomGen g, Random a):: (a, a) -> g -> (a, g)`, lo que significa que tiene comportamiento similar a `random`, solo que primero toma una dupla de valores que establecerán el límite superior e inferior de forma que el valor aleatorio generado esté dentro de ese rango.

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

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

También existe `randomRs`, la cual produce una secuencia de valores aleatorios dentro de nuestro rango.

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

Genial, tiene pinta de ser una contraseña de alto secreto.

Puedes estar preguntándote que tienes que ver esta sección con la E/S. Hasta ahora no hemos visto nada relacionado con la E/S. Bien, hasta ahora siempre hemos creado nuestro generador de forma manual basándonos en algún entero arbitrario. El problema es que, en los programas reales, siempre devolverán los mismos números aleatorios, lo cual no es muy buena idea. Por este motivo`System.Random` nos ofrece la acción de E/S `getStdGen` que tiene el tipo `IO StdGen`. Cuando se inicia la ejecución de un programa, éste pregunta al sistema por un buen generador de valores aleatorios y lo almacena en algo llamado generador global. `getStdGen` trae ese generador para que podamos ligarlo a algo.

Aquí tienes un programa simple que genera una cadena aleatoria.

```
import System.Random

main = do
    gen <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen)
```

```
$ runhaskell random_string.hs
pybphhzzhuepknbykxhe
$ runhaskell random_string.hs
eiqgcxykivpudlsvvjpg
$ runhaskell random_string.hs
nzdceoconysdgcyqjruo
$ runhaskell random_string.hs
bakzhnnuzrkgvesqplrx
```

Ten cuidad ya que al llamar dos veces a `getStdGen` estamos preguntándole dos veces al sistema por el mismo generador global. Si hacemos algo como:

```
import System.Random

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

Obtendremos la misma cadena mostrada dos veces. Una forma de obtener dos cadenas diferentes de 20 caracteres de longitud es crear una lista infinita y tomar los 20 primeros caracteres y mostrarlos en una línea, luego tomamos los 20 siguientes y los mostramos en una segunda línea. Para realizar esto podemos utilizar la función `splitAt` de `Data.List`, que divide una lista en un índice dado y devuelve una dupla que tiene la primera parte como primer componente y la segunda parte como segundo componente.

```
import System.Random
import Data.List

main = do
    gen <- getStdGen
    let randomChars = randomRs ('a','z') gen
        (first20, rest) = splitAt 20 randomChars
```

```
        (second20, _) = splitAt 20 rest
    putStrLn first20
    putStr second20
```

Otra forma de hacerlo es utilizando la acción `newStdGen` que divide el generador de valores aleatorios en dos nuevos generadores. Actualiza el generador global con uno de ellos y el toro lo de vuelve como resultado de la acción.

```
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 cuando ligamos `newStdGen`, sino que el generador global también se actualiza, así que si después utilizamos`getStdGen` obtendremos otro generador que será diferente a `gen`.

Vamos a crear un programa que haga que nuestro usuario adivine el número en el que estamos pensado.

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

main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        askForNumber newGen
```

Hemos creado la función `askForNumber`, que toma un generador de valores aleatorios y devuelve una acción de E/S que preguntará al usuario por un número y le dirá si ha acertado o no. Dentro de esta función, primero generamos un número aleatorio y nuevo generador basándonos en el generador que obtuvimos como parámetro, los llamamos `randNumber` y `newGen`. Digamos que el número generado es el `7`. Luego preguntamos al usuario en que número estamos pensando. Ejecutamos `getLine` y ligamos el resultado a`numberString`. Cuando el usuario introduce `7`, `numberString` se convierte en `"7"`. Luego, utilizamos una cláusula `when` para comprobar si la cadena que ha introducido el usuario está vacía. Si lo está, una acción de E/S vacía (`return ()`) se ejecutará, terminando así nuestro programa. Si no lo está, la acción contenida en el bloque `do` se ejecutará. Utilizamos`read` sobre `numberString` para convertirla en un número, el cual ahora será `7`.

Nota

Si el usuario introduce algo que `read` no pueda leer (como `"haha"`), nuestro programa terminará bruscamente con un mensaje de error bastante horrendo. Si no te apetece que el programa termine de esta forma, utiliza la función `reads`, que devuelve una lista vacía cuando no puede leer una cadena. Cuando si puede devuelve una lista unitaria que contiene una

dupla con nuestro valor deseado como primer componente y una cadena con lo que no ha consumido como segundo componente.

Comprobamos si el número que han introducido es igual al número que hemos generado aleatoriamente y damos al usuario un mensaje apropiado. Luego llamamos a `askForNumber` de forma recursiva, solo que esta vez con el nuevo generador que hemos obtenido, de forma que obtenemos una acción de E/S como la que acabamos de ejecutar, solo que depende de un generador diferente.

`main` consiste básicamente en obtener el generador de valores aleatorio y llamar a `askForNumber` con el generador inicial.

¡Aquí tienes nuestro programa en acción!

```
$ runhaskell guess_the_number.hs
Which number in the range from 1 to 10 am I thinking of? 4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of? 10
You are correct!
Which number in the range from 1 to 10 am I thinking of? 2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of? 5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

Otra forma de hacer el mismo programa sería:

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

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStr "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        newStdGen
        main
```

Es muy similar a la versión anterior, solo que en lugar de hacer una función que tome un generador y luego se llame a si misma de forma recursiva, hacemos todo el trabajo en `main`. Después de decir al usuario si el número que pensaba es correcto o no, actualizamos el generador global y volvemos a llamar a `main`. Ambas implementaciones son válidas pero a mi me gusta más la primera ya que el `main` realiza menos acciones y también nos proporciona una función que podemos reutilizar.

Cadenas de bytes

Las listas son unas estructuras de datos estupendas además útiles. Hasta ahora las hemos utilizado en cualquier sitio. Hay una multitud de funciones que operan con ellas y la evaluación perezosa de Haskell nos permite intercambiarlas por los bucles a la hora de realizar filtrados y trazados, ya que la evaluación solo ocurre cuando realmente se necesita, de modo que las listas infinitas (¡incluso listas infinitas de listas infinitas!) no son un problema para nosotros. Por este motivo las listas también se pueden utilizar para representar flujos de datos, ya sea para leer desde la entrada estándar o desde un fichero. Podemos abrir un fichero y leerlo

como si se tratase de una cadena, incluso aunque solo se acceda hasta donde alcancen nuestras necesidades.

Sin embargo, procesar ficheros como cadenas tiene un inconveniente: suele ser lento. Como sabes, `String` es sinónimo de tipo de `[Char]`. `Char` no tiene un tamaño fijo, ya que puede tomar varios bytes para representar un carácter. Ademas, las listas son perezosas. Si tienes un lista como`[1,2,3,4]`, se evaluará solo cuando sea completamente necesario. Así que la lista entera es una especie de promesa de que en algún momento será una lista. Recuerda que `[1,2,3,4]` es simplemente una decoración sintáctica para `1:2:3:4:[]`. Cuando el primer elemento de la lista es forzado a evaluarse (digamos que mostrándolo por pantalla), el resto de la lista`2:3:4:[]` sigue siendo una promesa de una lista, y así continuamente. Así que puedes pensar en las listas como si se tratasen de promesas de que el el siguiente elemento será entregado una vez sea necesario. No hace falta pensar mucho para concluir que procesar una simple lista de números como una serie de promesas no de debe ser la cosa más eficiente del mundo.

Esta sobrecarga no nos suele preocupar la mayor parte del tiempo, pero si debería hacerlo al la hora de leer y manipular ficheros de gran tamaño. Por esta razón Haskell posee **cadenas de bytes**. Las cadenas de bytes son una especie de listas, solo que cada elemento tiene el tamaño de un byte (o 8 bits). La forma en la que son evaluadas es también diferente.

Existen dos tipos de cadenas de bytes: las estrictas y las perezosas. Las estrictas residen en `Data.ByteString` y no posee ninguna evaluación perezosa. No hay ninguna promesa involucrada, un cadena de bytes estricta representa una serie de bytes en un vector. No podemos crear cosas como cadenas de bytes infinitas. Si evaluamos el primer byte de un cadena de bytes estricta evaluamos toda la cadena. La ventaja es que hay menos sobrecarga ya que no implica ningún *thunk* (término técnico de *promesa*). La desventaja es que consumirán memoria mucho más rápido ya que se leen en memoria de un solo golpe.

El otro tipo de cadenas de bytes reside en `Data.ByteString.Lazy`. Son perezosas, pero no de la misma forma que las listas. Como ya hemos dicho, hay tantos *thunks* como elementos en una cadena normal. Este es el porqué de que sean lentas en algunas situaciones. Las cadenas de bytes perezosas toman otra enfoque, se almacenan en bloques de 64KB de tamaño. De esta forma, si evaluamos un byte en una cadena de bytes perezosa (mostrándolo por pantalla o algo parecido), los primeros 64KB serán evaluados. Luego de estos, solo existe una promesa de que los siguientes serán evaluados. Las cadenas de bytes perezosas son como una especie de lista de cadenas de bytes de 64KB. Cuando procesemos ficheros utilizando cadenas de bytes perezosas, los contenidos del fichero serán leídos bloque a bloque. Es genial ya que no llevará la memoria hasta sus límite y probablemente 64KB caben perfectamente en la memoria cache L2 de tu procesador.

Si miras la Documentación de `Data.ByteString.Lazy`, verás que exporta un montón de funciones que tienen el mismo nombre que las de `Data.List`, solo que en sus declaraciones de tipo tienen `ByteString` en lugar de `[a]` y `Word8` de la `a` de su interior. Las funciones con nombres similares se comportan prácticamente igual salvo que unas trabajan con listas y las otras con cadenas de bytes. Como importan nombres de funciones iguales, vamos a importarlas de forma cualificada en nuestro código y luego lo cargaremos en GHCi para jugar con con las cadenas de bytes.

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

`B` posee las cadenas de bytes perezosas mientras que `S` contiene las estrictas. Utilizaremos casi siempre la versión perezosa.

La función `pack` tiene un tipo `[Word8] -> ByteString`. Lo cual significa que toma una lista de bytes del tipo `Word8` y devuelve una`ByteString`. Puedes verlo como si tomara un lista, que es perezosa, y la hace menos perezosa, de forma que sigue siendo perezosa solo que a intervalos de 64KB.

¿Qué sucede con el tipo `Word8`? Bueno, es como `Int`, solo que tiene un rango mucho más pequeño, de 0 a 255. Representa un número de 8b. Y al igual que `Int`, es miembro de la clase `Num`. Por ejemplo, sabemos que el valor 5 es polimórfico ya que puede comportarse como cualquier tipo numeral. Bueno, pues también puede tomar el tipo `Word8`.

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

Como puede ver, normalmente no tienes que preocupar mucho del tipo `Word8`, ya que el sistema de tipos puede hacer que los números tomen ese tipo. Si tratas de utilizar un número muy grande, como 336, como un `Word8`, simplemente se truncará de forma binaria al valor 80.

Hemos empaquetado solo unos pocos valores dentro de una cadena de bytes, de forma que caben dentro de un mismo bloque (`Chunk`). El `Empty` es como`[]` para las listas.

`unpack` es la versión inversa de de `pack`. Toma una cadena de bytes y la convierte en una lista de bytes.

`fromChunks` toma una lista de cadenas de bytes estrictas y la convierte en una cadena de bytes perezosa. `toChunks` toma una cadena de bytes perezosa y la convierte en una estricta.

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

Esto es útil cuando tienes un montón de cadenas de bytes estrictas y quieres procesarlas eficientemente sin tener que unirlas en memoria en una más grande primero.

La versión de `:` para cadenas de bytes se conoce como `cons`. Toma un byte y una cadena de bytes y pone dicho byte al principio. Aunque es perezosa, generará un nuevo bloque para ese elemento aunque dicho bloque aún no este lleno. Por este motivo es mejor utilizar la versión estricta de `cons`, `cons'` , si vas a insertar un montón de bytes al principio de una cadena de bytes.

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
ghci> B.cons' 85 $ B.pack [80,81,82,84]
Chunk "UPQRT" Empty
ghci> foldr B.cons B.empty [50..60]
Chunk "2" (Chunk "3" (Chunk "4" (Chunk "5" (Chunk "6" (Chunk "7" (Chunk "8" (Chunk "9" (C
ghci> foldr B.cons' B.empty [50..60]
Chunk "23456789:;<" Empty
```

Como puedes ver `empty` crea una cadena de bytes vacía ¿Puedes ver las diferencias entre `cons` y `cons'`? Con ayuda de `foldr` hemos empezado con una cadena de bytes vacía y luego hemos recorrido la lista de números desde la derecha, añadiendo cada número al principio de la cadena de bytes. Cuando utilizamos `cons`, acabamos con un bloque por cada byte, lo cual no es muy útil para nuestros propósitos.

De cualquier modo, los módulo de cadenas de bytes tienen un montón de funciones análogas a las de `Data.List`, incluyendo, pero no limitándose, a`head`, `tail`, `init`, `null`, `length`, `map`, `reverse`,`foldl`, `foldr`, `concat`, `takeWhile`, `filter`, etc.

También contienen funciones con el mismo nombre y comportamiento que algunas funciones que se encuentran en `System.IO`, solo que `String` se remplaza por `ByteString`. Por ejemplo, la función `readFile` de `System.IO`tiene el tipo

`readFile :: FilePath -> IO String`, mientras que `readFile` de los módulos de cadenas de bytes tiene el tipo `readFile :: FilePath -> IO ByteString`. Ten cuidado, si estás utilizando la versión estricta de cadenas de bytes e intentas leer un fichero, se leerá en memoria de un solo golpe. Con las cadenas de bytes perezosas se leerá bloque a bloque.

Vamos a crear un programa simple que tome dos rutas de ficheros como parámetros de la línea de comandos y copie el contenido del primero en el segundo. Ten en cuenta que `System.Directory` ya contiene una función llamada `copyFile`, pero vamos a implementar nuestro programa así de todas formas.

```
import System.Environment
import qualified Data.ByteString.Lazy as B

main = do
    (fileName1:fileName2:_) <- getArgs
    copyFile fileName1 fileName2

copyFile :: FilePath -> FilePath -> IO ()
copyFile source dest = do
    contents <- B.readFile source
    B.writeFile dest contents
```

Creamos nuestra propia función que toma dos `FilePath` (recuerda, `FilePath` es solo un sinónimo de `String`) y devuelve una acción de E/S que copiará el contenido de un fichero utilizando cadenas de bytes. En la función `main`, simplemente obtenemos los parámetros y llamamos a nuestra función con ellos para obtener una acción de E/S que será ejecutada.

```
$ runhaskell bytestringcopy.hs something.txt ../../something.txt
```

Fíjate que un programa que no utilice cadenas de bytes puede tener el mismo parecido, la única diferencia sería que en lugar de escribir `B.readFile` y `B.writeFile` usaríamos `readFile` y `writeFile`. Muchas veces podemos convertir un programa que utilice cadenas a un programa que utilice cadenas de bytes simplemente utilizando los módulos correctos y cualificando algunas funciones. A veces, pueden necesitar convertir funciones que trabajan con cadenas para que funcionen con cadenas de bytes, pero no es demasiado difícil.

Siempre que necesites un mayor rendimiento en programas que lean montones de datos en forma de cadenas, intenta utilizar cadenas de bytes, tendrás grandes posibilidades de conseguir un rendimiento mayor con muy poco esfuerzo. Normalmente yo suelo crear programas que trabajan con cadenas normales y luego las convierto a cadenas de bytes de el rendimiento no se ajusta a los objetivos.

Excepciones

Todos los lenguajes tienen procedimientos, funciones o trozos de código que fallan de alguna forma. Es una ley de vida. Lenguajes diferentes tienen formas diferentes de manejar estos fallos. En *C*, solemos utilizar un valor de retorno anormal (como -1 o un puntero nulo) para indicar que el valor devuelto no debe ser tratado de forma normal. *Java* y *C#*, por otra parte, tienden a utilizar excepciones para controlar estos fallos. Cuando se lanza una excepción, la ejecución de código salta a algún lugar que hemos definido para realice las tareas apropiadas e incluso quizá relance la excepción para que sea tratada en otro lugar.

Haskell tiene un buen sistema de tipos. Los tipos de datos algebraicos nos permiten tener tipos como `Maybe` y `Either` que podemos utilizar para representar resultados que son válidos y que no lo son. En *C*, devolver, digamos -1, cuando suceda un error es una cuestión de convención. Solo

tiene un significado especial para los humanos. Si no tenemos cuidado, podemos tratar esos datos anormales como válidos de forma que nuestro código termine siendo un auténtico desastre. El sistema de tipos de Haskell nos da la seguridad que necesitamos en este aspecto. Una función `a -> Maybe b` indica claramente que puede producir un `b` envuelto por un `Just` o bien puede devolver `Nothing`. El tipo es completamente diferente a `a -> b` y si intentamos utilizar estas dos funciones indistintamente, el sistema de tipos se quejará.

Aunque aún teniendo tipos expresivos que soporten operaciones erróneas, Haskell sigue teniendo soporte para excepciones, ya tienen más sentido en el contexto de la E/S. Un montón de cosas pueden salir mal cuando estamos tratando con el mundo exterior ya que no es muy fiable. Por ejemplo, cuando abrimos un fichero, bastantes cosas pueden salir mal. El fichero puede estar protegido, puede no existir o incluso que no exista un soporte físico para él. Así que está bien poder saltar a algún lugar de nuestro código que se encargue de un error cuando dicho error suceda.

Vale, así que el código de E/S (es decir, código impuro) puede lanzar excepciones. Tiene sentido ¿Pero qué sucede con el código puro? Bueno, también puede lanzar excepciones. Piensa en las funciones `div` y `head`. Tienen los tipos `(Integral a) => a -> a ->` y `[a] -> a` respectivamente. No hay ningún `Maybe` ni `Either` en el tipo que devuelven pero aun así pueden fallar. `div` puede fallar si intentas dividir algo por cero y`head` cuando le das una lista vacía.

```
ghci> 4 `div` 0
*** Exception: divide by zero
ghci> head []
*** Exception: Prelude.head: empty list
```

El código puro puede lanzar excepciones, pero solo pueden ser capturadas en las partes de E/S de nuestro código (cuando estamos dentro de un bloque `do`que es alcanzado por `main`). Esto ocurre así porque no sabemos cuando (o si) algo será evaluado en el código puro ya que se evalúa de forma perezosa y no tiene definido un orden de ejecución concreto, mientras que las partes de E/S sí lo tienen.

Antes hablábamos de como debíamos permanecer el menor tiempo posible en las partes de E/S de nuestro programa. La lógica de nuestro programa debe permanecer mayoritariamente en nuestras funciones puras, ya que sus resultados solo dependen de los parámetros con que las llamemos. Cuando tratas con funciones puras, solo tenemos que preocuparnos de que devuelve una función, ya que no puede hacer otra cosa. Esto hace nuestra vida más sencilla. Aunque realizar algunas tareas en la parte E/S es fundamental (como abrir un fichero y cosas así), deben permanecer al mínimo. Las funciones puras son perezosas por defecto, lo que significa que no sabemos cuando serán evaluadas y realmente tampoco nos debe preocupar. Sin embargo, cuando las funciones puras empiezan a lanzar excepciones, si importa cuando son evaluadas. Por este motivo solo podemos capturar excepciones lanzadas desde código puro en las partes de E/S de nuestro programa. Y como queremos mantener las partes de E/S al mínimo esto no nos beneficia mucho. Sin embargo, si no las capturamos en una parte de E/S de nuestro código, el programa se abortará ¿Solución? No mezcles las excepciones con código puro. Toma ventaja del potente sistema de tipos de Haskell y utiliza tipos como `Either` y `Maybe` para representar resultados que pueden ser erróneos.

Por este motivo, por ahora solo veremos como utilizar las excepciones de E/S. Las excepciones de E/S ocurren cuando algo va mal a la hora de comunicamos con el mundo exterior. Por ejemplo, podemos tratar de abrir un fichero y luego puede ocurrir que ese fichero ha sido eliminado o algo parecido. Fíjate en el siguiente programa, el cual abre un fichero que ha sido obtenido como parámetro y nos dice cuantas líneas contiene.

```
import System.Environment
import System.IO
```

```
main = do (fileName:_) <- getArgs
          contents <- readFile fileName
          putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
```

Un programa muy simple. Realizamos la acción de E/S `getArgs` y ligamos la primera cadena de la cadena que nos devuelve a `fileName`. Luego llamamos a los contenidos de fichero como `contents`. Para terminar, aplicamos`lines` a esos contenidos para obtener una lista de lineas y luego obtenemos la longitud de esa lista y la mostramos utilizando `show`. Funciona de la forma esperada, pero ¿Qué sucede cuando le damos el nombre de un fichero que no existe?

```
$ runhaskell linecount.hs i_dont_exist.txt
linecount.hs: i_dont_exist.txt: openFile: does not exist (No such file or directory)
```

¡Ajá! Obtenemos un error de *GHC* que nos dice que ese fichero no existe. Nuestro programa falla ¿Qué pasaría si quisiéramos mostrar un mensaje más agradable en caso de que el fichero no exista? Una forma de hacerlo sería comprobando si el fichero existe antes de intentar abrirlo utilizando la función `doesFileExist` de `System.Directory`.

```
import System.Environment
import System.IO
import System.Directory

main = do (fileName:_) <- getArgs
          fileExists <- doesFileExist fileName
          if fileExists
              then do contents <- readFile fileName
                      putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"
              else do putStrLn "The file doesn't exist!"
```

Hicimos `fileExists <- doesFileExist fileName` porque `doesFileExist`tiene como declaración de tipo `doesFileExist :: FilePath -> IO Bool`, lo que significa que devuelve una acción de E/S que tiene como resultado un valor booleano que nos dice si el fichero existe o no. No podemos utilizar`doesFileExist` directamente en una expresión `if`.

Otra solución sería utilizando excepciones. Es perfectamente aceptable utilizarlas en este contexto. Un fichero que no existe es una excepción que se lanza desde la E/S, así que capturarla en la E/S es totalmente aceptable.

Para tratar con esto utilizando excepciones, vamos a aprovecharnos de la función `catch` de `System.IO.Error`. Su declaración de tipo es`catch :: IO a -> (IOError -> IO a) -> IO a`. Toma dos parámetros. El primero es una acción de E/S. Por ejemplo, podría ser una acción que trate de abrir un fichero. El segundo es lo que llamamos un manipulador. Si la primera acción de E/S que le pasemos a `catch` lanza un excepción, la excepción pasa al manipulador que decide que hacer. Así que el resultado final será una acción que o bien actuará como su primer parámetro o bien hará lo que diga el manipulador en caso de que la primera acción de E/S lance una excepción.

Si te es familiar los bloques *try-catch* de lenguajes como *Java* o *Python*, la función `catch` es similar a ellos. El primer parámetro es lo que hay que intentar hacer, algo así como lo que hay dentro de un bloque *try*. El segundo parámetro es el manipulador que toma una excepción, de la misma forma que la mayoría de los bloques *catch* toman excepciones que puedes examinar para ver que ha ocurrido. El manipulador es invocado si se lanza una excepción.

El manipulador toma un valor del tipo `IOError`, el cual es un valor que representa que ha ocurrido una excepción de E/S. También contienen información acerca de la excepción que ha sido lanzada. La

implementación de este tipo depende de la implementación del propio lenguaje, por lo que no podemos inspeccionar valores del tipo `IOError` utilizando el ajuste de patrones sobre ellos, de la misma forma que no podemos utilizar el ajuste de patrones con valores del tipo `IO algo`. Sin embargo, podemos utilizar un montón de predicados útiles para examinar los valores del tipo `IOError` como veremos en unos segundos.

Así que vamos a poner en uso a nuestro nuevo amigo `catch`.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e = putStrLn "Whoops, had some trouble!"
```

Lo primero de todo, puedes ver como hemos utilizado las comillas simples para utilizar esta función de forma infija, ya que toma dos parámetros. Utilizarla de forma infija la hace mas legible. Así que `toTry `catch` handler` es lo mismo que `catch toTry handler`, además concuerda con su tipo. `toTry` es una acción de E/S que intentaremos ejecutar y `handler` es la función que toma un `IOError` y devuelve una acción que será ejecutada en caso de que suceda una excepción.

Vamos a probarlo:

```
$ runhaskell count_lines.hs i_exist.txt
The file has 3 lines!

$ runhaskell count_lines.hs i_dont_exist.txt
Whoops, had some trouble!
```

No hemos comprobado que tipo de `IOError` obtenemos dentro de `handler`. Simplemente decimos `"Whoops, had some trouble!"` para cualquier tipo de error. Capturar todos los tipos de excepciones un mismo manipulador no es una buena práctica en Haskell ni en ningún otro lenguaje ¿Qué pasaría si se lanzara alguna otra excepción que no queremos capturar, como si interrumpimos el programa o algo parecido? Por esta razón vamos a hacer lo mismo que se suele hacer en otros lenguajes: comprobaremos que tipo de excepción estamos capturando. Si la excepción es del tipo que queremos capturar, haremos nuestro trabajo. Si no, relanzaremos esa misma excepción. Vamos a modificar nuestro programa para que solo capture las excepciones debidas a que un fichero no exista.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | otherwise = ioError e
```

Todo permanece igual excepto el manipulador, el cual hemos modificado para que capture únicamente un grupo de excepciones de E/S. Hemos utilizado dos nuevas funciones de `System.IO.Error`, `isDoesNotExistError` y `ioError`.

`isDoesNotExistError` es un predicado sobre `IOError`, o lo que es lo mismo, es una función que toma un valor del tipo `IOError` y devuelve `True` o `False`, por lo que su declaración de tipo es`isDoesNotExistError :: IOError -> Bool`. Hemos utilizado esta función con la excepción que se le pasa al manipulador para ver si el error fue debido a que no existía un fichero. Utilizamos también la sintaxis de*guardas*, aunque podríamos haber utilizado un `if else`. En caso de que la excepción no fuera lanzada debido a que no se encuentre un fichero, relanzamos la excepción que se le pasó al manipulador utilizando la función `ioError`. Su declaración de tipo es `ioError :: IOException ->IO a`, así que toma un `IOError` y produce un acción de E/S que lanza esa excepción. La acción de E/S tiene el tipo `IO a` ya que realmente nunca devolverá un valor.

Resuminedo, si la excepción lanzada dentro de la acción de E/S `toTry` que hemos incluido dentro del bloque `do` no se debe a que no exista un fichero,`toTry `catch` handler` capturará esa excepción y la volverá a lanzar.

Existen varios predicados que trabajan con `IOError` que podemos utilizar junto las guardas, ya que, si una guarda no se evalua a `True`, se seguirá evaluando la siguiente guarda. Los predicados que trabajan con `IOError` son:

- `isAlreadyExistsError`
- `isDoesNotExistError`
- `isAlreadyInUseError`
- `isFullError`
- `isEOFError`
- `isIllegalOperation`
- `isPermissionError`
- `isUserError`

La moyoría de éstas se explican por si mismas. `isUserError` se evalua a`True` cuando utilizamos la función `userError` para crear la excepción, lo cual se utiliza para crear excepciones en nuestro código y acompañarlas con una cadena. Por ejemplo, puedes utilizar algo como`ioError $ userError "remote computer unplugged!"`, aunque es preferible que utilices los tipos `Either` y `Maybe` para representar posibles fallos en lugar de lanzar excepciones por ti mismo con `userError`.

Podríamos tener un manipulador que se pareciera a algo como esto:

```
handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e = putStrLn "The file doesn't exist!"
    | isFullError e = freeSomeSpace
    | isIllegalOperation e = notifyCops
    | otherwise = ioError e
```

Donde `notifyCops` y `freeSomeSpace` son acciones de E/S que hemos definido. Asegurate de relanzar las excepciones que no cumplan tu criterio, de lo contrario harás que tu programa falle de forma sigilosa cuando no debería.

`System.IO.Error` también exporta algunas funciones que nos permiten preguntar a estas excepciones por algunos atributos, como qué manipulador causó el error, o qué ruta de fichero lo provocó. Estas funciones comienzan por `ioe` y puedes ver la lista completaen la documentación. Digamos que queremos mostrar la ruta de un fichero que provocó un error. No podemos mostrar el `fileName` que obtuvimos de `getArgs`, ya que solo un valor del tipo `IOError` se pasa al manipulador y manipulador no sabe nada más. Una función depende exclusivamente de los parámetros con los que fue llamada. Por esta razón podemos utilizar la función `ioeGetFileName`, cuya declaración de tipo es `ioeGetFileName ::IOError -> Maybe FilePath`. Toma un `IOError` como parámetro y quizá devuelva un `FilePath` (que es un sinónimo de `String`, así que es prácticamente lo mismo). Básicamente lo que hace es extraer la ruta de un fichero de un `IOError`, si puede. Vamos a modificar el programa anterior para que muestre la ruta del fichero que provocó una posible excepción.

```
import System.Environment
import System.IO
import System.IO.Error

main = toTry `catch` handler

toTry :: IO ()
toTry = do (fileName:_) <- getArgs
           contents <- readFile fileName
           putStrLn $ "The file has " ++ show (length (lines contents)) ++ " lines!"

handler :: IOError -> IO ()
handler e
    | isDoesNotExistError e =
        case ioeGetFileName e of Just path -> putStrLn $ "Whoops! File does not exist at:
                                 Nothing -> putStrLn "Whoops! File does not exist at unkn
    | otherwise = ioError e
```

Si la guarda donde se encuentra `isDoesNotExistError` se evalua a `True`, utilizamos una expresión `case` para llamar a `ioeGetFileName` con `e`y aplicamos un ajuste de patrones con el `Maybe` que devuelve. Normalmente utilizamos las expresiones `case` cuando queremos aplicar un ajuste de patrones sin tener que crear una nueva función.

No tienes porque utilizar un manipulador para capturar todas las excepciones que ocurran en la parte de E/S de tu programa. Puedes cubrir ciertas partes de tu código de E/S con `catch` o puedes cubrir varias de ellas con `catch`y utilizar diferentes manipuladores. Algo como:

```
main = do toTry `catch` handler1
          thenTryThis `catch` handler2
          launchRockets
```

Aquí, `toTry` utiliza `handler1` como manipulador y `thenTryThis`utiliza `handler2`. `launchRockets` no es ningún parámetro de ningún`catch`, así que cualquier excepción que lanze abortará el programa, a no ser que `launchRockets` utilice internamente un `catch` que gestione sus propias excepciones. Por supuesto `toTry`, `thenTryThis` y`launchRockets` son acciones de E/S que han sido unidas con un bloque `do`e hipotéticamente definidas en algún lugar. Es similar a los bloques*try-catch* que aparecen en otro lenguajes, donde puedes utilizar un solo bloque *try-catch* para envolver a todo el programa o puede utilizar un enfoque más detallado y utilizar bloques diferentes en diferentes partes del programa.

Ahora ya sabes como tratar las excepciones de la E/S. No hemos visto como lanzar excepciones desde código puro y trabajar con ellas, porque, como ya hemos dicho, Haskell ofrece mejores formas de informar de errores sin recurrir a partes de la E/S. Incluso aun teniendo que trabajar con acciones de la E/S que puede fallar, prefiero tener tipos como `IO (Either a b)`, que indiquen que son acciones de E/S normales solo que su resultado será del tipo`Either a b`, así que o bien devolverán `Left a` o `Right b`.

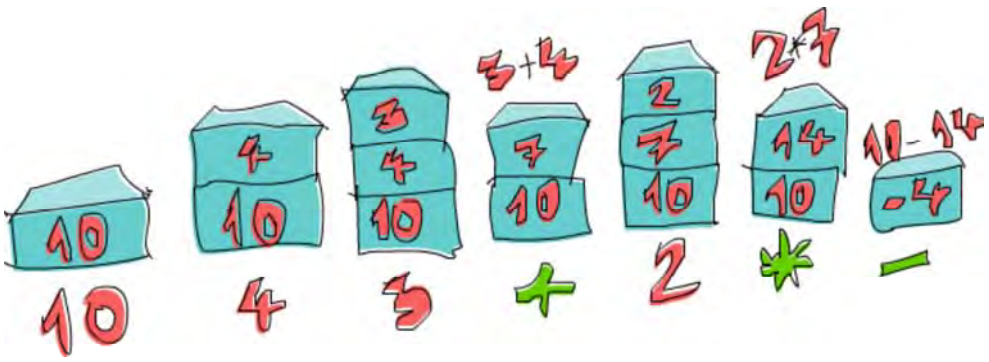
Resolviendo problemas de forma funcional

En este capítulo, veremos un par de problemas interesantes y como resolverlos de forma funcional y elegante. Probablemente no introduciremos ningún concepto nuevo, solo vamos a practicar nuestras habilidades de programación y calentar un poco. Cada sección presentará un problema diferente. Primero describiremos el problema, luego intentaremos resolverlo y trataremos de encontrar la mejor (o al menos no la peor) forma de resolverlo.

Notación polaca inversa

Normalmente cuando escribíamos expresiones matemáticas en la escuela lo hacíamos de forma infija. Por ejemplo, `10 - (4 + 3) * 2`. `+`, `*` y `-` son operadores infijos, al igual que los funciones infijas que conocemos de Haskell (`+`, *elem*, etc.). Resulta bastante útil, ya que nosotros, como humanos, podemos analizar fácilmente estas expresiones. La pega es que tenemos que utilizar paréntesis para especificar la precedencia.

La Notación polaca inversaes otra forma de escribir expresiones matemáticas. Al principio parece un poco enrevesado, pero en realidad es bastante fácil de entender y utilizar ya que no hay necesidad de utilizar paréntesis y muy fácil de utilizar en la calculadoras. Aunque las calculadoras más modernas usan una notación infija, todavía hay gente que lleva calculadoras RPN (del inglés, *Reverse PolishNotation*). Así se vería la expresión infija anterior en RPN:`10 4 3 + 2 * -` ¿Cómo calculamos el resultado de esto? Bueno, piensa en una pila. Recorremos la expresión de izquierda a derecha. Cada vez que encontramos un número, lo apilamos. Cuando encontramos un operador, retiramos los dos números que hay en la cima de la pila, utilizamos el operador con ellos y apilamos el resultado de nuevo. Cuando alcancemos el final de la expresión, debemos tener un solo número en la pila si la expresión estaba bien formada, y éste representa el resultado de la expresión



¡Vamos a realizar la operación `10 4 3 + 2 * -` juntos! Primero apilamos`10` de forma que ahora nuestra pila contiene un `10`. El siguiente elemento es un `4`, así que lo apilamos también. La pila ahora contiene`10, 4`. Hacemos los mismo para el `3` y conseguimos una pila que contiene`10, 4, 3`. Ahora, encontramos un operador, `+`. Retiramos los dos números que se encuentran en la cima de la pila (de forma que la pila se quedaría de nuevo solo con `10`), sumamos esos dos números y apilamos el resultado. La pila contiene `10, 7` ahora mismo. Apilamos `2` y obtenemos `10, 7, 2`. Multiplicamos `7` y `2` y obtenemos `14`, así que lo apilamos y la pila ahora contendrá `10, 14`. Para terminar hay un `-`. Retiramos `10` y`14` de la pila, restamos `14` a `10` y apilamos el resultado. El número que contiene la pila es `-4` y como no hay más números ni operadores en la expresión, ese es el resultado.

Ahora que ya sabemos como calcular una expresión RPN a mano, vamos a pensar en como podríamos hacer que una función Haskell tomara como parámetro una cadena que contenga una expresión RPN, como `10 4 3 + 2 * -`, y nos devolviera el resultado.

¿Cuál sería el tipo que debería tener esta función? Queremos que tome una cadena y produzca un número como resultado. Así que lo más seguro es que el tipo sea algo como `solveRPN :: (Num a) => String -> a`.

Nota

Ayuda mucho pensar primero en cual será la declaración de tipo de una función antes de preocuparnos en como implementarla para luego escribirla. Gracias al sistema de tipos de Haskell, la declaración de tipo de una función nos da mucha información acerca de ésta.

Bien. Cuando implementemos la solución de un problema en Haskell, a veces es bueno volver a ver como lo solucionamos a mano para ver si podemos sacar algo que nos ayude. En este caso vimos que tratábamos cada número u operador que estaba separado por un espacio como un solo elemento. Así que podría ayudarnos si empezamos rompiendo una cadena como `"10 4 3 + 2 * -"` en una lista de elementos como `["10","4","3","+","2","*","-"]`.

A continuación ¿Qué fue lo siguiente que hicimos mentalmente? Recorrimos la expresión de izquierda a derecha mientras manteníamos una pila ¿Te recuerda la frase anterior a algo? Recuerda la sección que hablaba de los*pliegues*, dijimos que cualquier función que recorra una lista de izquierda a derecha, elemento a elemento, y genere (o acumule) un resultado (ya sea un número, una lista, una pila o lo que sea) puede ser implementado con un pliegue.

En este caso, vamos a utilizar un pliegue por la izquierda, ya que vamos a recorrer la lista de izquierda a derecha. Nuestro acumulador será la pila, y por la tanto el resultado será también una pila, solo que, como ya hemos visto, contendrá un solo elemento.

Una cosa más que tenemos que pensar es, bueno ¿Cómo vamos a representar la pila? Propongo que utilicemos una lista. También propongo que mantengamos en la cabeza de la lista la cima de la pila. De esta forma añadir un elemento en la cabeza de la lista es mucho más eficiente que añadirlo al final. Así que si tenemos una pila como, `10, 4, 3`, la representaremos con una lista como`[3,4,10]`.

Ahora tenemos suficiente información para bosquejar vagamente nuestra función. Tomará una cadena como `"10 4 3 + 2 * -"` y la romperá en una lista de elementos utilizando `words` de forma que obtenga`["10","4","3","+","2","*","-"]`. Luego, utilizará un pliegue por la izquierda sobre esa lista y generará una pila con un único elemento, como`[-4]`. Tomará ese único elemento de la lista y ese será nuestro resultado final.

Aquí tienes el esqueleto de esta función:

```
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN expression = head (foldl foldingFunction [] (words expression))
    where   foldingFunction stack item = ...
```

Tomamos una expresión y la convertimos en una lista de elementos. Luego plegamos una función sobre esta lista. Ten en cuenta que `[]` representa es acumulador inicial. Dicho acumulador es nuestra pila, así que `[]` representa la pila vacía con la que

comenzamos. Luego de obtener la pila final que contiene un único elemento, llamamos a `head` sobre esa lista para extraer el elemento y aplicamos `read`.

Solo nos queda implementar la función de pliegue que tomará una pila, como`[4,10]` y un elemento, como `"3"` y devolverá una nueva pila`[4,10,3]`. Si la pila es `[4,10]` y el elemento es `"*"`, entonces tenemos que devolver `[40]`. Pero antes, vamos a transformar nuestra función al *estilo libre de puntos* ya que tiene muchos paréntesis y me está dando grima.

```
import Data.List

solveRPN :: (Num a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction stack item = ...
```

Ahí lo tienes. Mucho mejor. Como vemos, la función de pliegue tomará una pila y un elemento y devolverá una nueva pila. Utilizaremos ajuste de patrones para obtener los elementos de la cima de la pila y para obtener los operadores, como `"*"` o `"-"`.

```
solveRPN :: (Num a, Read a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction xs numberString = read numberString:xs
```

Hemos utilizado cuatro patrones. Los patrones se ajustarán de arriba a abajo. Primero, la función de pliegue verá si el elemento actual es `"*"`. Si lo es, tomará una lista como podría ser `[3,4,9,3]` y llamará a sus dos primeros elementos `x` e `y` respectivamente. En este caso, `x``sería``3` e `y` sería `4`. `ys` sería `[9,3]`. Devolverá una lista como`ys`, solo que tendrá `x` por `y` como cabeza. Con esto retiramos los dos elementos superiores de la pila, los multiplicamos y apilamos el resultado de nuevo en la pila. Si el elemento no es `"*"`, el ajuste de patrones fallará y continuará con `"+"`, y así sucesivamente.

Si el elemento no es ninguno de los operadores, asumimos que es una cadena que representa un número. Simplemente llamamos a `read` sobre esa esa cadena para obtener el número y devolver la misma pila pero con este número en la cima.

¡Y eso es todo! Fíjate que hemos añadido una restricción de clase extra (`read a`) en la declaración de la función, ya que llamamos a `read` sobre la cadena para obtener un número. De esta forma la declaración dice que puede devolver cualquier tipo que forme parte de las clases de tipos `Num` y`Read` (como `Int`, `Float`, etc.).

Para la lista de elementos `["2", "3", "+"]`, nuestra función empezará plegando la lista desde la izquierda. La pila inicial será `[]`. Llamará a la función de pliegue con `[]` como pila (acumulador) y `"2"` como elemento. Como dicho elemento no es un operador, utilizará `read` y añadirá el número al inicio de `[]`. Así que ahora la pila es `[2]` y la función de pliegue será llamada con `[2]` como pila y `"3"` como elemento, produciendo una nueva pila `[3,2]`. Luego, será llamada por tercera vez con`[3,2]` como pila y con `"+"` como elemento. Esto hará que los dos números sean retirados de la pila, se sumen, y que el resultado sea apilado de nuevo. La pila final es `[5]`, que contiene el número que devolveremos.

Vamos a jugar con esta función:

```
ghci> solveRPN "10 4 3 + 2 * -"
-4
ghci> solveRPN "2 3 +"
5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037
```

```
ghci> solveRPN "90 3 -"
87
```

¡Genial, funciona! Un detalle de esta función es que se puede modificar fácilmente para que soporte nuevos operadores. No tienen porque ser operadores binarios. Por ejemplo, podemos crear el operador `"log"` que solo retira un número de la pila y apila su logaritmo. También podemos crear operadores ternarios que retiren tres números de la pila y apilen un resultado, o operadores como `sum` que retiraría todos los números de la pila y devolvería su suma.

Vamos a modificar nuestra función para que acepte unos cuantos operadores más. Para simplificar, vamos a cambiar la declaración de tipo de forma que devuelva un número del tipo `Float`.

```
import Data.List

solveRPN :: String -> Float
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction (x:y:ys) "/" = (y / x):ys
            foldingFunction (x:y:ys) "^" = (y ** x):ys
            foldingFunction (x:xs) "ln" = log x:xs
            foldingFunction xs "sum" = [sum xs]
            foldingFunction xs numberString = read numberString:xs
```

¡Perfecto! ‘’/’’ es la división y ** la potencia de número en coma flotante. Con el operador logarítmico, usamos el ajuste de patrones para obtener un solo elemento y el resto de la lista, ya que solo necesitamos un elemento para obtener su logaritmo neperiano. Con el operador`sum`, devolvemos una pila con un solo elemento, el cual es la suma de toda la pila.

```
ghci> solveRPN "2.7 ln"
0.9932518
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

Fíjate que podemos incluir números en coma flotante en nuestra expresión porque `read` sabe como leerlos.

```
ghci> solveRPN "43.2425 0.5 ^"
6.575903
```

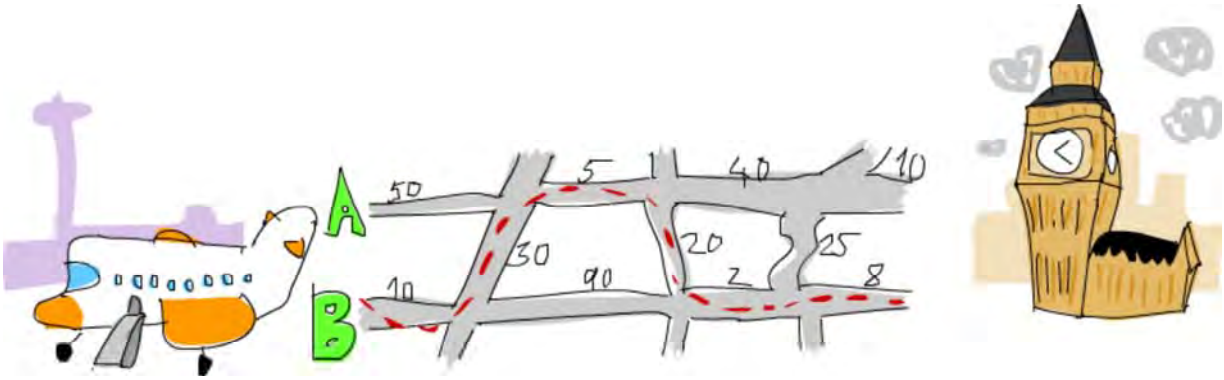
En mi opinión, crear una función que calcule expresiones arbitrarias RPN en coma flotante y tenga la opción de ser fácilmente extensible en solo 10 líneas es bastante impresionante.

Una cosa a tener en cuenta es que esta función no es tolerante a fallos. Cuando se da una entrada que no tiene sentido, simplemente bloqueará todo. Crearemos una versión tolerante a fallos de esta función con una declaración de tipo como `solveRPN :: String -> Maybe Float` una vez conozcamos las mónadas (no dan miedo, créeme). Podríamos crear una función como esta ahora mismo, pero sería un poco pesado ya que requeriría un montón de comprobaciones para `Nothing` en cada paso. Si crees que puede ser un reto, puedes continuar e intentarla crearla tu mismo. Un consejo: puedes utilizar `reads`para ver si una lectura a sido correcta o no.

De Heathrow a Londres

Nuestro siguiente problema es este: tu avión acaba de aterrizar en Inglaterra y alquilas un coche. Tienes una entrevista dentro de nada y tienes que llegar desde el aeropuerto de Heathrow a Londres tan pronto como puedas (¡Pero si arriesgar tu vida!).

Existen dos vías principales de Heathrow a Londres y hay cierto número de carreteras regionales que unen ambas vías. Debes encontrar la ruta óptima que te lleve a Londres tan rápido como puedas. Empiezas en el lado izquierdo y puedes o bien cruzar a la otra vía o continuar recto.



Como puedes ver en la imagen, la ruta más corta de Heathrow a Londres en este caso es empezando en la vía principal B, cruzamos y continuamos por A, cruzamos otra vez y continuamos dos veces más por B. Si tomamos esta ruta, tardaremos 75 minutos en llegar. Si tomamos cualquier otra ruta, tardaríamos más en llegar.

Nuestro trabajo es crear un programa que tome una entrada que represente un sistema de caminos y muestre cual es la ruta más corta. Así se vería la entrada para este caso.

```
50
10
30
5
90
20
40
2
25
10
8
0
```

Para analizar mentalmente el fichero de entrada, separa los números en grupos de tres. Cada grupo se compone de la vía A, la vía B y un camino que los une. Para que encajen perfectamente en grupos de tres, diremos que hay un último camino de cruce que recorrerlo toma cero minutos. Esto se debe a que no nos importa a que parte de Londres lleguemos, mientras lleguemos a Londres.

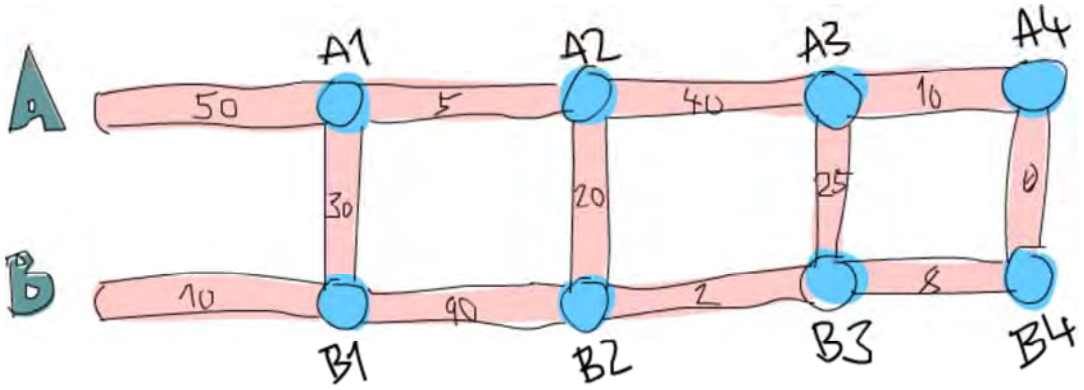
De la misma forma que solucionamos el problema de la calculadora RPN, este problema lo resolveremos en tres pasos:

- Olvida Haskell por un instante y piensa como solucionarías el problema a mano.
- Piensa como vamos a representar la información en Haskell.
- Encuentra un modo de operar sobre esta información en Haskell que produzca una solución.

En el problema de la calculadora, primero nos dimos cuenta de que cuando calculábamos una expresión a mano, manteníamos una especie de pile en nuestra cabeza y recorríamos la expresión elemento a elemento. Decidimos utilizar una lista de cadenas para representar la expresión. Para terminar, utilizamos un pliegue por la izquierda para recorrer la lista de cadenas mientras manteníamos un pila que producía una solución.

Vale ¿Cómo encontraríamos la ruta más corta de Heathrow a Londres a mano? Bueno, podemos simplemente ver todas las rutas y suponer cual será la más corta y seguramente sea verdad. Esa solución funciona bien para problemas pequeños, pero ¿Qué pasaría si las vías tuvieran más de 10.000 secciones? Ni siquiera podríamos dar una solución optima.

Así que no es una buena solución. Aquí tienes una imagen simplificada del sistema de caminos:



Esta bien ¿Puedes decir cual sería la ruta más corta hasta la primera intersección (El primer punto en A, marcado como A1) de la vía A? Es bastante trivial. Simplemente vemos si es mas corto ir recto desde A o si es más corto partir desde B y luego cruzar a la otra vía. Obviamente, es más corto ir por B y luego cruzar ya que toma 40 minutos, mientras que ir directamente desde A toma 50 minutos ¿Qué pasaría si quisiéramos ir a B1? Lo mismo. Vemos que es mucho más corto ir directamente desde B (10 minutos), ya que ir desde A y luego cruzan nos llevaría un total de 80 minutos.

Ahora sabemos la ruta más corta hasta A1 (ir desde la vía B y cruzar, diríamos algo como que es `B, C` con un coste de 40) y sabemos cual es la ruta más corta hasta B1 (ir directamente desde la vía B, simplemente `B` con coste 10) ¿Nos ayudaría en algo esta información si queremos saber la ruta más corta hasta la próxima intersección de ambas vías principales? ¡Por supuesto que sí!

Vamos a ver cual sería la ruta más corta hasta A2. Para llegar a A2, o bien iríamos directamente desde A1 o bien partiríamos desde B1, continuaríamos adelante y luego cruzaríamos (recuerda, solo podemos ir adelante o cruzar al otro lado). Y como sabemos el coste de A1 y B1, podemos encontrar fácilmente cual será la ruta más corta de A1 a A2. Costaría 40 minutos llegar a A1 y luego otros 5 minutos en llegar desde A1 a A2, así que el resultado sería `B,C, A` con un coste de 45 minutos. Solo cuesta 10 minutos llegar hasta B1, pero luego costaría otros 110 minutos más para llegar hasta A2. Así que, de forma bastante obvia, la forma más rápida de llegar a A2 es `B, C, A`. Del mismo modo, la forma más rápida de llegar hasta B2 es continuar por A1 y luego cruzar.

Nota

¿Qué pasaría si para llegar a A2 primero cruzamos desde B1 y luego continuamos adelante? Bien, ya hemos cubierto la posibilidad de cruzar de B1 a A1 cuando buscábamos la mejor forma de llegar hasta A1, así que no tenemos que tomar en cuenta esta posibilidad en el siguiente paso.

Ahora que tenemos la mejor ruta para llegar hasta A2 y B2, podemos repetir este proceso indefinidamente hasta que alcancemos el final. Una vez tengamos las mejores rutas para llegar a A4 y B4, la mejor será la ruta óptima.

En el segundo paso básicamente hemos repetido lo que hicimos en el primer paso, solo que tuvimos en cuenta cuales fueron las mejores rutas para llegar a A y B. También podríamos decir que tomamos en cuenta las mejores rutas para llegar hasta A y B en el primer paso, solo que ambas rutas tendrían coste 0.

Así que en resumen, para obtener las mejor ruta ruta de Heathrow a Londres, hacemos esto: primero vemos cual es la mejor ruta hasta el próximo cruce de la vía principal A. Las dos opciones que tenemos son o bien ir directamente o bien empezar en la vía opuesta, continuar adelante y luego cruzar. Memorizamos la mejor ruta y el coste. Usamos el mismo método para ver cual es la mejor ruta hasta el próximo cruce desde B y la memorizamos. Luego, vemos si la ruta del siguiente cruce en A es mejor si la tomamos desde el cruce anterior en A o desde el cruce anterior en B y luego cruzar. Memorizamos la mejor ruta y hacemos lo mismo para la vía opuesta. Repetimos estos pasos hasta que alcancemos el final. La mejor de las dos rutas resultantes será la ruta óptima.

Básicamente lo que hacemos es mantener la mejor ruta por A y la mejor ruta por B hasta que alcancemos el final, y la mejor de ambas es el resultado. Sabemos como calcular la ruta más corta a mano. Si tuviéramos suficiente tiempo, papel y lápiz, podríamos calcular la ruta más corta de un sistema de caminos con cualquier número de secciones.

¡Siguiente paso! ¿Cómo representamos este sistema de caminos con los tipos de datos de Haskell? Una forma es ver los puntos iniciales y las intersecciones como nodos de un grafo que se conectan con otras intersecciones. Si imaginamos que los nodos iniciales en realidad se conectan con cada otro nodo con un camino, veríamos que cada nodo se conecta con el nodo del otro lado y con el nodo siguiente del mismo lado. Exceptuando los nodos finales, que únicamente se conectan con el nodo del otro lado.

```
data Node = Node Road Road | EndNode Road
data Road = Road Int Node
```

Un nodo es o bien un nodo normal que contiene información acerca del camino que lleva al otro nodo de la otra vía principal o del camino que lleva al siguiente nodo, o bien un nodo final, que solo contiene información acerca del camino que lleva al otro nodo de la otra vía principal. Un camino contiene la información que indica lo que se tarda en recorrerlo y el nodo al que lleva. Por ejemplo, la primera parte del camino de la vía A sería `Road 50 a1`donde `a1` sería un nodo `Node x y`, donde `x` e `y` serían los caminos a B1 y a A2.

Otra forma de representar el sistema sería utilizando `Maybe` para los caminos que llevan al siguiente nodo. Cada nodo tendría un camino que llevara a otro punto de la vía opuesta, pero solo los nodos que no están al final tendrían un camino que les llevará adelante.

```
data Node = Node Road (Maybe Road)
data Road = Road Int Node
```

Ambas son buenas formas de representar el sistema de caminos en Haskell y en realidad podríamos resolver el problema usándolas, pero, quizá podemos encontrar algo más simple. Si pensamos de nuevo en la forma de resolverlo a mano, vemos que en realidad siempre comprobamos los tiempos de los tres caminos de una sección a la vez: la parte del camino en la vía A, la parte opuesta en B y la parte C, que conecta ambas entre sí. Cuando estábamos buscando la ruta más corta entre A1 y B1, solo tuvimos que tratar con los tiempos de las primeras tres partes, los cuales eran 50, 10 y 30 minutos. Nos referiremos a esto como una sección. Así que el sistema de caminos que utilizamos para este ejemplo puede representarse fácilmente como cuatro secciones: `50, 10, 30`, `5, 90, 20`, `40, 2, 25` y `10, 8, 0`.

Siempre es bueno mantener nuestros tipos de datos tan simple como sea posible, pero ¡No más simple!

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int } deriving (Show)
type RoadSystem = [Section]
```

¡Es casi perfecto! Es simple y tengo la sensación de que va a funcionar perfectamente para la implementación de nuestra solución. `Section` es un tipo de dato algebraico simple que contiene tres enteros para los tiempos de los tres caminos de una sección. También hemos utilizado un sinónimo de tipo que dice que `RoadSystem` es una lista de secciones.

Nota

También podríamos haber utilizado una tripla como`(Int, Int, Int)` para representar una sección. Está bien utilizar tuplas en lugar de tipos de datos algebraicos propios para cosas pequeñas y puntuales, pero normalmente es mejor crear nuevos tipos para cosas como esta. De esta forma el sistema de tipos tiene más infomación acerca de que es cada cosa. Podemos utilizar`(Int, Int, Int)` para representar una sección de un camino o para representar un vector en un espacio tridimensional y podemos trabajar con ambos a la vez, pero de este modo podríamos acabar mezclandolos entre sí. Si utilizamos los tipos `Section` y`Vector`, no podremos, ni si quiera accidentalmente, sumar un vector a una sección.

Ahora el sistema de caminos de Heathrow a Londres se puede representar así:

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [Section 50 10 30, Section 5 90 20, Section 40 2 25, Section 10 8 0]
```

Todo lo que nos queda por hacer es implementar la solución a la que llegamos con Haskell ¿Cual sería la declaración de tipo de una función que calcule el camino más corto para cualquier sistema de caminos? Tendría que tomar un sistema de caminos y devolver una ruta. Vamos a representar una ruta con una lista también. Crearemos el tipo `Label` que será una simple enumeración cuyos valores serán `A`, `B` o `C`. También crearemos un sinónimo de tipo: `Path`.

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

Llamaremos a nuestra función `optimalPath` y tendrá una declaración de tipo como `optimalPath :: RoadSystem -> Path`. Si es llamada con el sistema`heathrowToLondon` deberá devolver una ruta como:

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

Vamos a tener que recorrer la lista de secciones de izquierda a derecha y mantener un camino óptimo hasta A y un camino óptimo hasta B conforme vayamos avanzando. Acumularemos la mejor ruta conforme vayamos avanzando, de izquierda a derecha ¿A qué te suena esto? ¡Ding, ding, ding! ¡Correcto, es un pliegue por la izquierda!

Cuando resolvimos el problema a mano, había un paso que repetíamos una y otra vez. Requería comprobar el camino óptimo de A y B hasta el momento, además de la sección actual para producir un nuevo par de rutas óptimas hasta A y B. Por ejemplo, al principio la rutas óptimas son `[]` y `[]` para A y B. Analizamos la sección `Section 50 10 30` y concluimos que la nueva ruta óptima para A es `[(B,10),(C,30)]` y que la nueva ruta óptima para B es`[(B,10)]`. Si vemos este paso como una función, tomaría un par de rutas y una sección y produciría un nuevo par de rutas. El tipo sería`(Path, Path) -> Section -> (Path, Path)`. Vamos a seguir adelante e implementar esta función que parece que será útil.

Nota

Será util porque `(Path, Path) -> Section -> (Path, Path)` puede ser utilizado como una función binaría para un pliegue por la derecha, el cual tiene un tipo `a -> b -> a`.

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let priceA = sum $ map snd pathA
        priceB = sum $ map snd pathB
        forwardPriceToA = priceA + a
        crossPriceToA = priceB + b + c
        forwardPriceToB = priceB + b
        crossPriceToB = priceA + a + c
        newPathToA = if forwardPriceToA <= crossPriceToA
                     then (A,a):pathA
```

```
                    else (C,c):(B,b):pathB
        newPathToB = if forwardPriceToB <= crossPriceToB
                    then (B,b):pathB
                    else (C,c):(A,a):pathA
    in  (newPathToA, newPathToB)
```

¿Qué hace esto? Primero, calculamos el coste óptimo en la vía A basandonos en el camino óptimo hasta el momento en A, y luego hacemos lo mismo para B. Hacemos `sum $ map snd pathA`, así que si `pathA` es algo como`[(A,100),(C,20)]`, `priceA` será `120`. `forwardPriceToA` es el coste de que tendría continuar hasta el siguiente cruce si fuéramos directamente desde el cruce anterior en A. Es igual al coste anterior de A, más el coste de la parte A de la sección actual. `crossPriceToA` es el coste que tendría si fuéramos hasta el siguiente cruce de A partiendo de B y luego cruzáramos. Este coste sería el coste óptimo de llegar al anterior cruce de B más el coste de continuar por B más el coste de cruzar por C. Calculamos`forwardPriceToB` y `crossPriceToB` de la misma forma.

Ahora que sabemos el mejor camino hasta A y B, solo tenemos que crear nuevas rutas para llegar hasta la siguiente intersección de A y B basándonos en estos. Si tardamos menos en llegar partiendo de A y continuando adelante, establecemos `newPathToA` a `(A,a):pathA`. Básicamente añadimos `LabelA` y el coste de la sección `a` al camino óptimo de A hasta el momento. Dicho de otro modo, decimos que la mejor forma de llegar al siguiente cruce de A es la ruta de llegar al cruce de A anterior y luego continuando adelante por la vía A. Recuerda que es `A` es una simple etiqueta, mientras que `a`tiene el tipo `Int` ¿Por qué añadimos el nuevo elemento al inicio en lugar de hacer algo como `pathA ++ [(A,a)]`? Bueno, añadir un elemento al principio de una lista es mucho más rápido que añadirlo al final. De este modo la ruta estará invertida cuando terminemos el pliegue con esta función, pero podemos invertirla de nuevo luego. Si tardamos menos en llegar al siguiente cruce de A partiendo del cruce anterior en B y luego cruzando, entonces`newPathToB` será la ruta anterior por B, continuar adelante y cruzar a A. Hacemos lo mismo para `newPathToB`, solo que al revés.

Terminamos devolviendo `newPathToA` y `newPathToB` en una tupla.

Vamos a ejecutar esta función con la primera sección de `heathrowToLondon`. Como es la primera sección, las mejores rutas hasta A y B serán un par de listas vacías.

```
ghci> roadStep ([], []) (head heathrowToLondon)
([(C,30),(B,10)],[(B,10)])
```

Recuerda que las rutas están invertidas, así que léelas de derecha a izquierda. Podemos ver que la mejor ruta hasta el siguiente cruce en A es empezando por B y luego cruzar hasta A y que la mejor ruta hasta B es simplemente continuando adelante a partir de B.

Nota

Cuando hacemos `priceA = sum $ map snd pathA`, estamos calculando el coste de la ruta en cada paso. No tendríamos que hacerlo si implementamos `roadStep` como una función `(Path, Path, Int, Int) -> Section -> (Path, Path, Int, Int)`donde los enteros representan el coste de A y B.

Ahora que tenemos una función que toma un par de rutas y una sección y produce una nueva ruta óptima, podemos hacer fácilmente un pliegue por la izquierda de la lista de secciones. `roadStep` se llamará con `([],[])` y la primera sección y devolverá una dupla con las rutas óptimas para esa sección. Luego será llamada con esa dupla de rutas y la sección siguiente y así sucesivamente. Cuando hayamos recorrido todas las secciones, tendremos una dupla con las rutas óptimas, y la mas corta será nuestra respuesta. Tendiendo esto en cuenta, podemos implementar `optimalPath`.

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
    in  if sum (map snd bestAPath) <= sum (map snd bestBPath)
            then reverse bestAPath
            else reverse bestBPath
```

Plegamos `roadSystem` por la izquierda (recuerda, es una lista de secciones) con un acumulador inicial que es una dupla de listas vacías. El resultado de ese pliegue es una dupla de rutas, así que usamos un ajuste de patrones sobre ella y obtenemos las rutas. Luego, comprobamos cual de esas dos es mejor y la devolvemos. Antes de devolverla, la invertimos, ya que las rutas óptimas están al revés debido a que decidimos añadir las secciones al principio de las listas.

¡Vamos a probarla!

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

¡Este es el resultado que se supone que debíamos obtener! ¡Genial! Se diferencia un poco del resultado que esperábamos ya que hay un paso `(C,0)`al final, lo que significa que tomamos un cruce cuando ya estamos en Londres, pero como tomar dicho camino no cuesta nada, sigue siendo la solución correcta.

Ahora que ya tenemos la función que encuentra la ruta óptima, solo tenemos que leer la representación textual del sistema de caminos por la entrada estándar, convertirlo en el tipo `RoadSystem`, ejecutar `optimalPath`sobre él y mostrar el resultado.

Antes de nada, vamos a crear una función que tome una lista y la divida en grupos del mismo tamaño. La llamaremos `groupsOf`. Con un parámetro como`[1..10]`, `groupsOf 3` deberá devolver`[[1,2,3],[4,5,6],[7,8,9],[10]]`.

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

Una función recursiva estándar. Para un `xs` de `[1..10]` y un `n` de`3`, equivale a `[1,2,3] : groupsOf 3 [4,5,6,7,8,9,10]`. Cuando la recursión termina, obtenemos una lista de grupos de tres elementos. Y aquí esta la función `main`, la cual leer desde la entrada estándar, crea un `RoadSystem` y muestra la ruta más corta:

```
import Data.List

main = do
    contents <- getContents
    let threes = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathPrice = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "The price is: " ++ show pathPrice
```

Primero, obtenemos todos los contenidos de la entrada estándar. Luego llamamos a `lines` con los contenidos para convertir algo como `"50\n10\n30\n..."`en `["50","10","30"...` y luego mapeamos `read` sobre ella para obtener una lista de números.

También llamamos a `groupsOf 3` sobre ella de forma que obtengamos una lista de listas de longitud tres. Mapeamos la función lambda `(\[a,b,c] -> Section a b c)` sobre esta lista de listas. Como puedes ver, esta función lambda toma una lista de tamaño tres y devuelve una sección. Así que `roadSystem` es nuestro sistema de caminos e incluso tiene el tipo correcto, `RoadSystem` (o `[Section]`). Llamamos `optimalPath` sobre éste y mostramos la ruta y el coste de la ruta óptima que obtenemos.

Guardamos el siguiente texto:

```
50
10
30
5
90
20
40
2
25
10
8
0
```

En un fichero llamado `paths.txt` y luego se lo pasamos a nuestro programa.

```
$ cat paths.txt | runhaskell heathrow.hs
The best path to take is: BCACBBC
The price is: 75
```

¡Funciona perfecto! Puedes usar tu conocimiento del módulo `Data.Random` para generar un sistema de caminos mucho más grande, que luego podrás pasar a nuestro programa de la misma forma que hemos hecho. Si obtienes errores de desbordamiento de pila, intenta usar `foldl'` en lugar `foldl`, ya que`foldl'` es estricto.

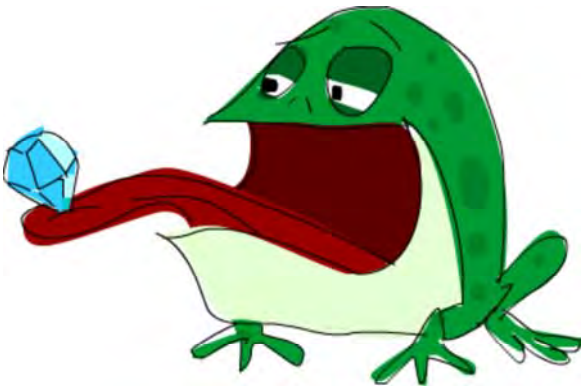
Funtores, funtores aplicativos y monoides

La combinación de Haskell de la pureza, las funciones de orden superior, tipos de datos algebraicos con parámetros, y clases de tipos nos permite implementar el polimorfismo a un nivel superior al que pueden alcanzar otros lenguajes. No tenemos que ver los tipos como si pertenecieran a una gran jerarquía de tipos. En lugar de eso, vemos como pueden actuar los tipos y luego los conectamos con las clases de tipos apropiadas. Un `Int` puede actuar como un montón de cosas. Puede actuar como algo equiparable, como algo ordenable, como algo enumerable, etc.

Las clases de tipos son abiertas, lo que significa que podemos definir nuestro propio tipo de dato, razonar en como éste actúa y conectarlo con la clase de tipos que define ese comportamiento. Por este motivo, y porque el fabuloso sistema de tipos de Haskell nos permite saber mucho acerca de una función con tan solo sabiendo su declaración de tipo, podemos crear clases de tipos que definen comportamientos muy generales y abstractos. Ya vimos que las clases de tipos definen operaciones para ver si dos cosas son iguales o comparar dos cosas por un cierto orden. Son comportamientos muy abstractos a la vez que elegantes, pero no los vemos como algo especial ya que hemos estado tratando con ellos a lo largo de nuestras vidas. Hace poco conocimos los funtores, que son básicamente cosas que se pueden mapear. Esto es un ejemplo de algo útil y a la vez bastante abstracto de lo que pueden describir las clases de tipos. En este capítulo veremos más de cerca los funtores, junto a una versión más fuerte y útil de los funtores llamados funtores aplicativos. También daremos un vistazo a los monoides.

De vuelta con los funtores

Ya hablamos de los funtores en su pequeña *sección*. Si todavía no la has leído, probablemente deberías darle un vistazo ahora mismo, o quizá luego cuando tengas más tiempo. O simplemente puedes hacer como si ya la hubieses leído.



Aun así, vamos a recordar algo: los funtores son cosas que se puede mapear, como las listas, `Maybe``s, árboles, etc. En Haskell, son descritos con laclase de tipos ``Functor`, la cual solo contiene un método de clase,`fmap`, que tiene como tipo `fmap :: (a -> b) -> f a -> f b`. Dice algo como: dame una función que tome un `a` y devuelva un `b` y una caja con una `a` (o varias de ellas) dentro y yo te daré una caja con una `b` (o varias de ellas) dentro. En cierto modo es como si aplicará la función dentro de la caja.

Nota

Muchas veces utilizamos la analogía de la caja para hacernos una idea de como funcionan los funtores, luego, probablemente usemos la misma analogía para los funtores aplicativos y las mónadas. Al principio es una buena analogía que ayuda a la gente a entender los funtores, pero no la tomes al pie de la letra, ya que para algunos funtores la analogía de la caja tiene que ser ajusta al milímetro para que siga siendo verdad. Un término más correcto para definir lo que es un funtor sería *contexto computacional*. El contexto sería que la computación podría tener un valor, o podría fallar (`Maybe` y `Either a`) o que podría tener más valores (listas) o cosas por el estilo.

Si queremos que un constructor de tipos sea una instancia de `Functor`, tiene que pertenecer a la familia de tipos `* -> *`, lo que significa que debe tomar exactamente un tipo concreto como parámetro. Por ejemplo, `Maybe`puede ser una instancia ya que tome un tipo como parámetro para producir un nuevo tipo concreto, como `Maybe Int` o `Maybe String`. Si un constructor de tipos toma dos parámetros, como `Either`, tenemos que aplicar parcialmente el constructor de tipos hasta que solo acepte un parámetro. Así que no podemos usar `instance Functor Either where` pero si podemos utilizar`instance Functor (Either a) where` y luego podemos pensar que `fmap` es solo para `Either a`, por lo que tendría una declaración de tipo como `fmap :: (b -> c) -> Either a b -> Either a c`. Como puedes ver, la parte`Either a` es fija, ya que `Either a` toma solo un parámetro, mientras que`Either` toma dos parámetros así que`fmap :: (b -> c) -> Either b -> Either c` no tendría mucho sentido.

Hasta ahora hemos aprendido como unos cuantos tipos (bueno, en realidad constructores de tipos) son instancias de `Functor`, como `[]`, `Maybe`, `Either a` y el tipo `Tree` que creamos nosotros mismos. Vimos como podíamos mapear funciones sobre ellos. En esta sección, veremos dos instancias más de la clase funtor, en concreto `IO` y `(->) r`.

Si un valor tiene el tipo, digamos, `IO String`, significa que es una acción que, cuando sea ejecutada, saldrá al mundo real y nos traerá una cadena, que será devuelta como resultado. Podemos usar `<-` dentro de un bloque `do`para ligar ese resultado a un nombre. Mencionamos que las acciones E/S son como cajas con sus pequeñitos pies que se encargan de salir al mundo real y traernos algún valor. Podemos inspeccionar lo que nos han traído, pero si lo hacemos el valor que devolvamos tiene que estar dentro de `IO`. Si pensamos en esta analogía de la caja con pies, podemos ver que `IO` se comporta como un funtor.

Vamos a ver como `IO` es una instancia de `Functor`. Cuando aplicamos`fmap` con una función sobre una acción de E/S, queremos obtener una acción de E/S que haga lo mismo, pero que tenga la función anterior aplicada a su resultado.

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

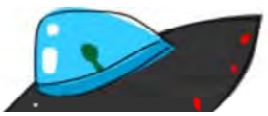
El resultado de mapear algo sobre una acción de E/S será una acción de E/S, así que justo después de la declaración usamos un bloque `do` para juntar dos acciones de E/S en una nueva. En la implementación de `fmap`, creamos una nueva acción de E/S que primero ejecutará la acción de E/S original y llamará a su resultado `result`. Luego, hacemos `return (f result)`.`return` es, como ya sabes, una función que crear una acción de E/S que no hace nada salvo tener algo como resultado. La acción que produce un bloque`do` siempre tendrá como resultado el resultado de su última acción. Por ese motivo utilizamos `return` para crear una acción de E/S que en realidad no hace nada, salvo contener `f result` como resultado.

Podemos jugar con él para ver como funciona. En realidad es bastante simple. Fíjate en el siguiente trozo de código:

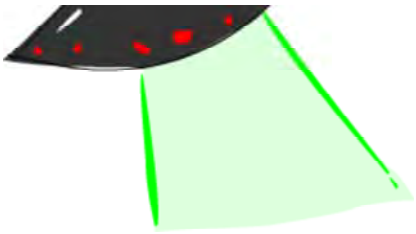
```
main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"
```

Se le pregunta al usuario por una nueva línea y luego se la devolvemos al usuario, aunque invertida. Así sería como escribiríamos lo mismo utilizando`fmap`:

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```



De la misma forma que que cuando usamos `fmap reverse` sobre `Just "blah"`obtenemos `Just "halb"`, podemos utilizar `fmap reverse` sobre`getLine`.



`getLine` es una acción de E/S que tiene el tipo `IO String`y al mapear `reverse` sobre ella nos devuelve una acción que viajará al mundo real y nos traerá una línea de texto, a la que luego dará la vuelta aplicando `reverse` a su resultado. De la misma forma que podemos aplicar una función a algo contenido en una caja `Maybe`, podemos aplicar una función a lo que hay dentro de una caja `IO`, solo que tiene que viajar al mundo real para obtener ese algo. Luego lo ligamos a un nombre usando `<-`,dicho nombre será asociado al resultado que ya se le ha aplicado `reverse`.

La acción de E/S `fmap (++"!") getLine` actúa como `getLine`, solo su resultado siempre lleva añadido un `"!"` al final.

Si vemos el tipo de `fmap` limitado a `IO`, tendríamos algo como `fmap :: (a -> b) -> IO a -> IO b`. `fmap` toma una función y una acción de E/S y devuelve una nueva acción de E/S que actúa como la anterior, solo que la función se aplica al resultado contenido en la acción.

Si alguna vez te encuentras ligando un nombre a una acción de E/S, con el único fin de aplicarle una función para luego usarlo en algún otro lugar, considera el uso de `fmap`, ya que es más elegante. Si quieres aplicar varias transformaciones al contenido de un funtor puedes declarar tu propia función, usar una función lambda o, idealmente, utilizar la composición de funciones:

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-' . reverse . map toUpper) getLine
          putStrLn line
```

```
$ runhaskell fmapping_io.hs
hello there
E-R-E-H-T- -O-L-L-E-H
```

Como probablemente ya sepas, `intersperse '-' . reverse . map toUpper` es una función que toma una cadena, mapea `toUpper` sobre ella, aplica`reverse` sobre el resultado anterior y luego le aplica `intersperse '-'`.Es como `(\xs -> intersperse '-' (reverse (map toUpper xs)))` solo que más bonito.

Otra instancia de `Functor` con la que hemos estado trabajando pero que no sabíamos que era un funtor es `(->) r`. Probablemente ahora mismo estás un poco confundido, ya que ¿Qué diablos significa `(->) r`? El tipo de una función `r -> a` se puede reescribir como `(->) r a`, de forma similar que podemos escribir `2 + 3` como `(+) 2 3`. Cuando nos encontramos con`(->) r a`, vemos a `(->)` de forma diferente, ya que podemos verlo como un constructor de tipos que toma dos parámetros de tipos, como `Either`. Pero recuerda, dijimos que un constructor de tipos debe tomar un solo parámetro para poder ser instancia de un funtor. Por esa razón no podemos crear una `(->)` instancia de `Functor`, pero si lo aplicamos parcialmente `(->) r`, no hay ningún problema. Si la sintaxis permitiera aplicar parcialmente los constructores de tipos con secciones (de la misma forma que podemos aplicar parcialmente `+` utilizando `(2+)`, que es lo mismo que `(+) 2`), podríamos escribir `(->) r` como `(r ->)` ¿Cómo son los funtores funciones? Bueno, vamos a echar un vistazo a la implementación, que se encuentra en `Control.Monad.Instances`.

Nota

Normalmente identificamos a la funciones que toman cualquier cosa y devuleven cualquier otra cosa como `a -> b`. `r -> a` es exactamente lo mismo, solo que hemos usado letras diferentes para las variables de tipo.

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

Si la sintaxis lo permitiera, lo podríamos haber escrito como:

```
instance Functor (r ->) where
    fmap f g = (\x -> f (g x))
```

Pero no lo permite, así que lo tenemos que escribir como al principio.

Antes de nada, vamos a pensar en el tipo de `fmap`. Sería `fmap :: (a -> b) -> f a -> f b`. Ahora lo que tenemos que hacer es remplazar mentalmente todas las `f`, las cuales hacen el papel de funtor, por `(->) r`. Hacemos esto cada vez que queramos ver como se comporta `fmap` para una cierta instancia. Obtenemos `fmap :: (a -> b) -> ((->) r a) -> ((->) r b)`. Ahora lo que podemos hacer es escribir los tipos `((->) r a)` y `((->) r b)` de forma infija, `r -> a` y `r -> b`, como hacemos normalmente con las funciones. Lo que obtenemos es `fmap :: (a -> b) -> (r -> a) -> (r -> b)`.

Mmmm... Vale. Si mapeamos una función sobre una función obtenemos una nueva función, de la misma forma que si mapeamos una función sobre un `Maybe` obtenemos un `Maybe` y de la misma forma que si mapeamos una función sobre una lista obtenemos una lista ¿Qué nos dice exactamente el tipo `fmap :: (a -> b) -> (r -> a) -> (r -> b)`? Bueno, podemos ver que toma una función de `a` a `b` y una función de `r` a `a` y devuelve una función de `r` a `b` ¿Te recuerda a algo? ¡Sí! ¡Composición de funciones! Dirigimos la salida de `r -> a` a la entrada de `a -> b` para obtener una función `r -> b`, lo cual es exactamente lo mismo que la composición de funciones. Si miras como se definió la instancia arriba, podrás ver que es una simple composición de funciones. Otra forma de escribirlo sería así:

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

De esta forma vemos de forma clara que `fmap` es simplemente una composición de funciones. Ejecuta `:m + Control.Monad.Instances`, ya que ahí está definida esta instancia e intenta mapear algunas funciones.

```
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
"300"
```

Podemos llamar a `fmap` de forma infija para que se parezca a `.`. En la segunda línea estamos mapeando `(*3)` sobre `(+100)`, lo que resulta en una función que tomara un valor llamará a `(+100)` y luego a `(*3)` con el resultado anterior. Llamamos a la función con `1`.

¿Cómo encaja la analogía de la caja aquí? Bueno, si la forzamos un poco, se ajusta. Cuando usamos `fmap (+3)` sobre `Just 3` nos es fácil imaginar a `Maybe` como una caja que contiene algo a lo que aplicamos la función `(+3)` ¿Pero qué sucede cuando usamos `fmap (*3) (+100)`? Bueno, puedes imaginar a `(+100)` como una caja que contiene el resultado final. Algo parecido a cuando imaginábamos las acciones de E/S como una caja que salía al mundo real y nos traía un resultado. Al usar `fmap (*3)` sobre `(+100)` creará otra función que se comportará como `(+100)`, solo que antes de producir el resultado, aplicará `(*3)` a ese resultado. Ahora podemos ver como `fmap` se comporta como `.` para las funciones.

El hecho de que `fmap` se comporte como una composición de funciones cuando se utiliza sobre funciones no es que sea especialmente útil en estos momentos, pero al menos es interesante. También puede confundirnos ver como algunas cosas que se comportan más como una computación que como una caja (`IO` y `(->) r`), son funtores. Una función mapeada sobre una computación devuelve esa misma computación, pero a el resultado de dicha computación se le aplicará la función mapeada.

Antes de que veamos las reglas que `fmap` debe seguir, vamos a pensar sobre el tipo de `fmap` una vez más. Su tipo es`fmap :: (a -> b) -> f a -> f b`. Nos hemos olvidado de la restricción de clase `(Functor f) =>`, pero lo hemos hecho por brevedad ya porque estamos hablando de funtores y sabemos que significa `f`. La primera vez que hablamos sobre las *Funciones currificadas*, dijimos que en realidad todas las funciones de Haskell toman un solo parámetro. Una función con tipo `a -> b -> c` en realidad toma un solo parámetro `a` y luego devuelve una función `b -> c`, que a su vez toma otro parámetro y devuelve`c`. Es como si llamáramos a la función con demasiados pocos parámetros (es decir, la aplicamos parcialmente), obtenemos una función que toma tantos parámetros como nos hayamos dejado (si pensamos de nuevo que las funciones toman varios parámetros). Así que `a -> b -> c` puede escribirse como`a -> (b -> c)` para hacer visible la currificación.

Del mismo modo, si escribimos `fmap :: (a -> b) -> (f a -> f b)`, podemos ver a `fmap` no como una función que toma una función y un funtor y devuelve otro funtor, sino como una función que toma una función y devuelve otra función igual a la anterior, solo que toma un funtor como parámetros y devuelve otro funtor como resultado. Toma una función `a -> b` y devuelve una función `f a -> f b`. A esto se llama *mover una función*. Vamos a trastear un poco con esa idea utilizando el comando `:t` de GHCi:

```
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]
```

La expresión `fmap (*2)` es una función que toma un funtor `f` sobre números y devuelve otro funtor sobre números. Ese funtor puede ser una lista, un `Maybe`, un `Either String`, cualquier cosa. La expresión`fmap (replicate 3)` tomara un funtor sobre cualquier tipo y devolverá un funtor sobre una lista de elementos de ese tipo.

Nota
Cuando decimos *funtor sobre números*, puedes verlo como un *funtor contiene números*. El primero es algo más formal y más técnicamente correcto, pero el segundo es más fácil de captar.

Puedes ver `fmap` como una función que toma una función y un funtor y luego mapea dicha función sobre el funtor, o puedes verlo como una función que toma función y mueve dicha función de forma que opere sobre funtores. Ambos puntos de vista son correctos en Haskell, equivalentes.

El tipo `fmap (replicate 3) :: (Functor f) => f a -> f [a]` nos dice que la función funcionará cuan cualquier tipo de funtor. Lo que hará exactamente dependerá de que tipo de funtor utilicemos. Si usamos `fmap (replicate 3)`con una lista, la implementación de `fmap` para listas será utilizada, que es `map`. Si la usamos con un``Maybe``, aplicará `replicate 3` al valor contenido en `Just`, o si es `Nothing`, devolverá `Nothing`.

```
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"
```

Ahora vamos a ver las **leyes de los funtores**. Para que algo sea una funtor, debe satisfacer una serie de leyes. Se espera que todos los funtores exhiban una serie de propiedades y comportamientos. Deben comportarse fielmente como cosas que se puedan mapear. Al llamar `fmap` sobre un funtor solo debe mapear una función sobre ese funtor, nada más. Este comportamiento se describe en las leyes de los funtores. Hay dos de ellas que todas las instancias de`Functor` deben cumplir. Haskell no comprueba estas leyes automáticamente, así que tenemos que comprobarlas nosotros mismos.

La primera ley de funtores establece que si mapeamos la función `id`sobre un funtor, el funtor que obtenemos debe ser igual que el original. Si lo escribimos algo más formal, sería `fmap id = id`. Básicamente dice que, si usamos `fmap id` sobre un funtor, debe devolver lo mismo que si aplicamos `id` a ese funtor. Recuerda, `id` es la función identidad, la cual devuelve el parámetro original que le pasemos. También se pude definir como `\x -> x`. Si vemos el funtor como algo que puede ser mapeado, la ley`fmap id = id` es bastante trivial y obvia.

Vamos a ver si esta ley se cumple para algunos funtores:

```
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
```

Si vemos la definición de `fmap` para, digamos, el tipo `Maybe`, podemos averiguar porque la primera ley se cumple:

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

Imaginamos que `if` hace el papel del parámetro `f` en la implementación. Vemos que si mapeamos `fmap id` sobre `Just x`, el resultado será`Just (id x)`, y como `id` simplemente devuelve su parámetro, podemos deducir que `Just (id x)` es igual a `Just x`. De esta forma ahora sabemos que si mapeamos `id` sobre un valor de `Maybe` con un constructor de datos`Just`, obtenemos lo mismo como resultado.

Demostrar que al mapear `id` sobre un valor `Nothing` devuelve el mismo valor es trivial. Así que a partir de estas dos ecuaciones de la implementación de `fmap` podemos decir que la ley `fmap id = id` se cumple.

La segunda ley dice que si mapeamos el resultado de una composición de dos funciones sobre un funtor debe devolver lo mismo que si mapeamos una de estas funciones sobre el funtor inicial y luego mapeamos la otra función. Escrito formalmente

sería `fmap (f . g) = fmap f . fmap g`. O de otra forma sería, para cualquier funtor `F`,`fmap (f . g) F = fmap f (fmap g F)`.

Si podemos demostrar que un funtor cumple las dos leyes, podemos confiar en que dicho funtor tendrá el mismo comportamiento que los demás funtores. Sabremos que cuando utilizamos `fmap` sobre él, no pasará nada más que no conozcamos y que se comportará como algo que puede ser mapeado, es decir, un funtor. Puedes averiguar si se cumple la segunda ley para cierto tipo viendo la implementación de `fmap` de ese tipo y utilizando luego el mismo método que hemos utilizado para ver si `Maybe` cumplía la primera ley.

Si quieres, podemos comprobar como se cumple la segunda ley de los funtores para `Maybe`. Si hacemos `fmap (f . g)` sobre `Nothing` obtenemos`Nothing`, ya que al mapear cualquier función sobre `Nothing` devuelve`Nothing`. Si hacemos `fmap f (fmap g Nothing)` sobre `Nothing`, obtenemos `Nothing` por el mismo motivo. Vale, comprobar como se cumple la segunda ley para `Maybe` si es un valor `Nothing` es bastante sencillo, casi trivial.

¿Qué pasa cuando tenemos un valor `Just algo`? Bueno, si hacemos`fmap (f . g) (Just x)`, a partir de la implementación vemos que convierte en `Just ((f . g) x)`, que es lo mismo que `Just (f (g x))`. Si hacemos`fmap f (fmap g (Just x))`, a partir de la implementación vemos que`fmap g (Just x)` es `Just (g x)`. Ergo, `fmap f (fmap g (Just x))` es igual a `fmap f (Just (g x))` y a partir de la implementación vemos que esto es igual a `Just (f (g x))`.

Si esta demostración te confunde un poco, no te preocupes. Asegúrate de entender como funciona la *composición de funciones*. La mayor parte de las veces puedes ver como se cumplen estas leyes de forma intuitiva porque los tipos actúan como contenedores o funciones. También puedes probarlas con cierta seguridad usando un montón de valores diferentes de un cierto tipo y comprobar que, efectivamente, las leyes se cumplen.

Vamos a ver un ejemplo patológico de un constructor de tipos que tenga una instancia de clase de tipos `Functor` pero que en realidad no sea un funtor, debido a que satisface las leyes. Digamos que tenemos el siguiente tipo:

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```

La *C* viene de *contador*. Es un tipo de datos que se parece mucho a`Maybe a`, solo que la parte `Just` contiene dos campos en lugar de uno. El primer campo del constructor de datos `CJust` siempre tiene el tipo`Int`, que es una especie de contador, mientras que el segundo campo tiene el tipo `a`, que procede del parámetro de tipo y su tipo será el tipo concreto que elijamos para `CMaybe a`. Vamos a jugar un poco con este nuevo tipo para ver como funciona.

```
ghci> CNothing
CNothing
ghci> CJust 0 "haha"
CJust 0 "haha"
ghci> :t CNothing
CNothing :: CMaybe a
ghci> :t CJust 0 "haha"
CJust 0 "haha" :: CMaybe [Char]
ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

Cuando usamos el constructor `CNothing`, no hay ningún campo que rellenar, mientras que si usamos el constructor `CJust`, el primer campo será un entero y el segundo campo podrá ser de cualquier tipo. Vamos a crear una instancia para la clase de

tipos `Functor` de forma que cada vez que usemos `fmap`, la función sea aplicada al segundo campo, mientras que el contador sea incrementado en uno.

```
instance Functor CMaybe where
    fmap f CNothing = CNothing
    fmap f (CJust counter x) = CJust (counter+1) (f x)
```

Se parece a la implementación de `Maybe`, exceptuando que cuando aplicamos `fmap` sobre un valor que no representa una caja vacía (un valor `CJust`), no solo aplicamos la función al contenido de la caja, sino que además incrementamos el contador en uno. Parece que todo está bien hasta ahora, incluso podemos probarlo un poco:

```
ghci> fmap (++"ha") (CJust 0 "ho")
CJust 1 "hoha"
ghci> fmap (++"he") (fmap (++"ha") (CJust 0 "ho"))
CJust 2 "hohahe"
ghci> fmap (++"blah") CNothing
CNothing
```

¿Cumple con las leyes de los funtores? Para demostrar que no cumple las leyes, basta con encontrar un contraejemplo.

```
ghci> fmap id (CJust 0 "haha")
CJust 1 "haha"
ghci> id (CJust 0 "haha")
CJust 0 "haha"
```

¡Ah! Sabemos que la primera ley de los funtores dice que si mapeamos `id` sobre un funtor, debe devolver lo mismo que llamamos `id` con ese mismo funtor, pero como hemos visto en este ejemplo, esta ley no se cumple para el funtor `CMaybe`. Aunque forma parte de la clase de tipos `Functor`, no cumple las leyes de los funtores y por lo tanto no es un funtor. Si alguien usará `CMaybe` como un funtor, esperaría que obedeciera las leyes de los funtores como un buen funtor. Pero `CMaybe` falla a la hora de ser un funtor aunque pretende serlo, así que usarlo como un funtor nos puede llevar a un código erróneo. Cuando utilizamos un funtor, no nos debe importar si primero unimos unas cuantas funciones usando una composición y luego la mapeamos sobre un funtor o si mapeamos unas cuantas funciones sobre un funtor sucesivamente. Pero con `CMaybe` si importa, ya que lleva una cuenta de cuantas veces ha sido mapeado ¡Mal! Si quisiéramos que `CMaybe` cumpliera las leyes de los funtores, deberíamos hacer que el campo `Int` se mantuviera constante utilizamos `fmap`.

En un principio las leyes de los funtores pueden parecer un poco confusas e innecesarias, pero luego vemos que si sabemos que un tipo cumple con ambas leyes, podemos asumir como se comportará. Si un tipo cumple las leyes de los funtores, sabemos que si llamamos a `fmap` sobre un valor de ese tipo solo mapeará la función sobre ese funtor, nada más. Esto nos lleva a un código que es más abstracto y extensible, ya que podemos utilizar las leyes para razonar acerca del comportamiento que un funtor debe tener y crear funciones que operen de forma fiable sobre funtores.

Todas las instancias de los funtores de la biblioteca estándar cumplen con estas leyes, aunque puedes comprobarlo tu mismo si no me crees. La próxima vez que hagas una instancia `Functor` para un tipo, tómate tu tiempo para asegurarte de que cumple con las leyes de los funtores. Cuando hayas trabajado lo suficiente con los funtores, sabrás ver de forma intuitiva las propiedades y comportamientos que tienen en común los funtores y te será muy fácil decir si un funtor cumple o no con estas leyes. Aún sin esta experiencia, siempre puedes leer la implementación línea a línea y ver si las leyes se cumplen o intentar descubrir algún contraejemplo.

También podemos ver los funtores como resultados en un cierto contexto. Por ejemplo, `Just 3` tiene un resultado igual a `3` en el contexto de que puede existir un resultado o no. `[1,2,3]` contiene tres resultados, `1`,`2` y `3`, en el contexto de que pueden haber varios resultados o incluso ninguno. La función `(+3)` dará un resultado, dependiendo del parámetro que se le de.

Si ves los funtores como cosas que puede producir resultados, puedes pensar que mapear algo sobre un funtor es como añadir una transformación al resultado de ese funtor que modificará el resultado. Cuando hacemos`fmap (+3) [1,2,3]`, añadimos la transformación `(+3)` al resultado`[1,2,3]`, de forma que cada vez que encuentre un número en la lista resultante, se le aplicará `(+3)`. Otro ejemplo sería mapear sobre funciones. Cundo hacemos `fmap (+3) (*3)`, añadimos la transformación `(+3)` al resultado final de `(*3)`. Verlo de este modo nos da un pista de porque al usar `fmap` sobre funciones equivale a componer funciones (`fmap (+3) (*3)` es igual a `(+3) . (*3)`, que equivale a`\x -> ((x*3)+3)`), ya que si tomamos una función como `(*3)` le añadimos la transformación `(+3)` a su resultado. Al final seguiremos teniendo una función, solo que cuando le demos un número, primero se multiplicará por tres y luego se le sumará tres, que es exactamente lo mismo que sucede con la composición de funciones.

Funtores aplicativos

En esta sección, daremos un vistazo a los funtores aplicativos, los cuales son una especie de funtores aumentados, representados en Haskell por la clase de tipos `Applicative`, que se encuentra en `Control.Applicative`.

Como ya sabes, las funciones en Haskell están currificadas por defecto, lo que significa que las funciones que parecen que toman varios parámetros en realidad solo toman un parámetro y devuelven una función que tomará el siguiente parámetro y así sucesivamente. Si una función tiene el tipo`a -> b -> c`, normalmente decimos que toma dos parámetros y devuelve un`c`, pero en realidad toma un `a` y devuelve una función `b -> c`. Por este motivo podemos aplicar esta función como `f x y` o como `(f x) y`. Este mecanismo es el que nos permite aplicar parcialmente las funciones simplemente pasándoles menos parámetros de los que necesitan, de forma que obtenemos nuevas funciones que probablemente pasaremos a otras funciones.

Hasta ahora, cuando mapeamos funciones sobre funtores, normalmente mapeamos funciones que toman un solo parámetro. Pero ¿Qué sucede si mapeamos una función como `*`, que toma dos parámetros, sobre un funtor? Vamos a ver varios ejemplo concretos. Si tenemos `Just 3` y hacemos`fmap (*) (Just 3)` ¿Qué obtenemos? A partir de la implementación de la instancia `Functor` de `Maybe`, sabemos que es un valor `Just algo`, aplicará la función `*` dentro de `Just`. Así pues, al hacer`fmap (*) (Just 3)` obtenemos `Just ((*) 3)`, que también puede escribirse usando secciones como `Just (* 3)` ¡Interesante! ¡Ahora tenemos una función dentro de un `Just`!

```
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 un tipo `(Ord a) => a -> a -> Ordering`sobre una lista de caracteres, obtenemos una lista de funciones del tipo`Char -> Ordering`, ya que la función `compare` se aplica parcialmente a cada uno de los caracteres de la lista. No es una lista de funciones`(Ord a) => a -> Ordering`, ya que como el primer `a` ha sido fijado a`Char` el segundo también debe ser `Char`.

Vemos que si aplicamos funciones con varios parámetros sobre funtores, obtenemos funtores que contienen funciones. Así que ¿Qué podemos hacer ahora con ellos? Bien, podemos mapear funciones que toman estas funciones como parámetros sobre ellos, ya que cualquier cosa que este dentro de un funtor será pasado a la función que mapeamos.

```
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 ¿Y si tenemos un valor funtor de `Just (3 *)` y un valor funtor de`Just 5` y queremos sacar la función de `Just (3 *)` y mapearla sobre`Just 5`? Con los funtores normales no tendríamos mucha suerte, ya que lo único que soportan es mapear funciones normales sobre funtores. Incluso aunque mapeáramos `\f -> f 9` sobre un funtor que contuviese funciones, estaríamos mapeando simples funciones normales. No podemos mapear funciones que están dentro de un funtor sobre otro funtor con lo que nos ofrece `fmap`. Podríamos usar un ajuste de patrones con el constructor `Just` para extraer la función y luego mapearla sobre `Just 5`, pero estamos buscando algo más abstracto, general y que funcione junto a los funtores.

Te presento la clase de tipos `Applicative`. Reside en el módulo`Control.Applicative` y define dos métodos, `pure` y `<*>`. No proporciona ninguna implementación por defecto para ninguno de los dos, así que tenemos que definir ambos si queremos que algo sea un funtor aplicativo. La clase se define así:

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

Estas tres simples líneas nos dicen mucho. Vamos a empezar por la primera línea. Empieza con la definición de la clase `Applicative` y también presenta una restricción de clase. Dice que si queremos que un constructor de tipos forme parte de la clase de tipos `Applicative`, tiene que ser primero parte de clase `Functor`. De este modo si sabemos que un constructor de tipos es parte de la clase de tipos `Applicative`, también lo es de`Functor`, así que podemos usar `fmap` sobre él.

El primer método que define se llama `pure`. Su declaración de tipo es`pure :: a -> f a`. `f` juega el papel del funtor aplicativo de la instancia. Como Haskell tiene un buen sistema de tipos y como todo lo que puede hacer una función es tomar un parámetro y devolver algún valor, podemos deducir muchas cosas únicamente a partir de la declaración de tipos, y este caso no es una excepción. `pure` debe tomar un valor de cualquier tipo y devolver un funtor aplicativo que contiene ese valor. Cuando decimos*que contiene*, estamos usando la analogía de la caja de nuevo, aunque ya hemos visto que esta comparación no siempre es perfecta. Aun así, la declaración `a -> f a` es bastante descriptiva. Tomamos un valor y lo introducimos en un funtor aplicativo que contendrá ese valor como resultado.

Una forma mejor de entender `pure` sería decir que toma un valor y lo introduce en una especie de contexto por defecto (o contexto puro), es decir, el contexto mínimo para albergar ese valor.

La función `<*>` es realmente interesante. Tiene como declaración de tipo`f (a -> b) -> f a -> f b` ¿Te recuerda a algo? Por supuesto,`fmap :: (a -> b) -> f a -> f b`. Es una especie de `fmap` modificado. Mientras que `fmap` toma una función y un funtor y aplica esa función dentro del funtor, mientras que `<*>` toma un funtor que contenga una función y otro funtor de forma que extrae esa función del primer funtor y la mapea sobre el segundo funtor. Cuando decimos `extrae`, en realidad es algo como ejecuta y luego extrae, quizá incluso secuenciar. Lo veremos pronto.

Vamos a echar un vistazo a la implementación de la instancia `Applicative` de `Maybe`.

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

De nuevo, a partir de la definición de la clase venos que `f` toma el papel funtor aplicativo que toma un tipo concreto como parámetro, así que escribimos `instance Applicative Maybe where` en lugar de`instance Applicative (Maybe a) where`.

Antes de nada, `pure`. Antes hemos dicho que se supone que éste toma algo y lo introduce en un funtor aplicativo. Hemos escrito `pure = Just`, ya que los constructores de datos como `Just` son funciones normales. También podríamos haber escrito `pure x = Just x`.

Luego, tenemos la definición de `<*>`. No podemos extraer una función de`Nothing`, ya que no hay nada dentro él. Así que decimos que si intentamos extraer una función de un `Nothing`, el resultado será `Nothing`. Si vemos la definición de clase de `Applicative`, veremos que hay una restricción de clase a `Functor`, lo cual significa que podemos asumir que ambos parámetros de `<*>` son funtores. Si el primer parámetro no es un `Nothing`, si no un`Just` con una función en su interior, diremos que queremos mapear esa función sobre el segundo parámetro. Esto también tiene en cuenta el caso en el que el segundo parámetro sea `Nothing`, ya que aplicar `fmap` con cualquier función sobre `Nothing` devuelve `Nothing`.

Así que para `Maybe`, `<*>` extrae la función de su operando izquierdo si es un `Just` y lo mapea sobre su operando derecho. Si alguno de estos parámetros es `Nothing`, `Nothing` será el resultado.

Vale, genial. Vamos a probarlo.

```
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
```

Vemos que tanto `pure (+3)` como `Just (3)` son iguales en este caso. Utiliza `pure` cuando trabajes con valores `Maybe` en un contexto aplicativo (es decir, cuando los utilices junto `<*>`), de cualquier otro modo sigue fiel a `Just`. Las primeras cuatro líneas de entrada demuestran como una función es extraída y luego mapeada, pero en este caso, podría haber sido logrado simplemente mapeando funciones normales sobre funtores. La última línea es interesante, ya que intentamos extraer una función de un `Nothing` y luego mapearla, lo cual es por supuesto `Nothing`.

Con los funtores normales solo podemos mapear una función sobre un funtor, luego no podemos extraer el resultado de forma general, incluso aunque el resultado sea una función parcialmente aplicada. Los funtores aplicativos, por otra parte, te permiten operar con varios funtores con una única función. Mira esto:

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

¿Qué esta pasando aquí? Vamos a echar un vistazo paso a paso. `<*>` es asociativo por la izquierda, por lo tanto `pure (+) <*> Just 3 <*> Just 5`es lo mismo que `(pure (+) <*> Just 3) <*> Just 5`. Primero, la función`+` se introduce en un funtor, en este caso un valor `Maybe` que contiene esa función. Así que al principio tenemos `pure (+)` que es lo mismo que`Just (+)`. Luego tenemos `Just (+) <*> Just 3`, cuyo resultado, debido a que se aplica parcialmente la función, es `Just (3+)`. Al aplicar `3` a la función `+` obtenemos una nueva función que tomará un parámetro y le añadirá 3. Para terminar, llegamos a `Just (3+) <*> Just 5`, que resulta en`Just 8`.

¿No es increíble? Los funtores aplicativos y el estilo aplicativo de hacer`pure f <*> x <*> y <*> ...` nos permiten tomar una función que espera parámetros que no son necesariamente funtores y utilizarla para operar con varios valores que están en algún contexto funtor. La función puede tomar tantos parámetros como queramos, ya que será aplicada parcialmente paso a paso cada vez que aparezca un `<*>`.

Todo esto se vuelve más útil y aparente si consideramos el hecho de que`pure f <*> x` es igual a `fmap f x`. Esta es una de la leyes aplicativas. Las veremos en detalle más adelante, pero por ahora, podemos ver de forma intuitiva su significado. Piensa un poco en ello, tiene sentido. Como ya hemos dicho, `pure` inserta un valor en un contexto por defecto. Si todo lo que hacemos es insertar una función en un contexto por defecto y luego la extraemos para aplicarla a un valor contenido en un funtor aplicativo, es lo mismo que simplemente mapear la función sobre ese funtor aplicativo. En lugar de escribir `pure f <*> x <*> y <*> ...` podemos usar`fmap f x <*> y <*> ...`. Por este motivo `Control.Applicative` exporta una función llamada `<$>`, que es simplemente `fmap` como operador infijo. Así se define:

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

Nota
Recuerda, las variables de tipo son independientes de los nombres de los parámetros o de otro nombres de valores. La `f` en la declaración de la función es una variable de tipo con una restricción de clase diciendo que cualquier constructor de tipos que reemplace a `f` de ser miembro de la clase `Functor`. La `f`que aparece en el cuerpo de la función representa la función que mapearemos sobre `x`. El hecho de que usemos `f` para representar ambos no significa que representen lo mismo.

El estilo aplicativo realmente destaca cuando utilizamos `<$>`, ya que si queremos aplicar un función `f` entre tres funtores aplicativos podemos escribirlo así `f <$> x <*> y <*> z`. Si los parámetros no fueran funtores aplicativos sino valores normales, lo habríamos escrito así`f x y z`.

Vamos a ver más de cerca como funciona. Tenemos un valor `Just "johntra"` y un valor `Just "volta"` y queremos unirlos en una sola `String` dentro de un funtor `Maybe`. Hacemos esto:

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

Antes de que veamos qué sucede aquí, compara lo anterior con esto:

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

¡Bien! Para usar una función normal con funtores aplicativos, simplemente tenemos que que esparcir unos cuantos `<$>` y `<*>` y la función operará sobre funtores aplicativos ¿No es genial?

De cualquier modo, cuando hacemos`(++) <$> Just "johntra" <*> Just "volta"`, primero `(++)`, que tiene un tipo `(++) :: [a] -> [a] -> [a]`, se mapea sobre `Just "johntra"`, lo cual da como resultado un valor `Just ("johntra"++)` cuyo tipo es`Maybe ([Char] -> [Char])`. Fíjate como el primer parámetro de `(++)` ha desaparecido y que `a` se ha convertido en un `Char`. Luego nos encontramos con `Just ("johntra"++) <*> Just "volta"`, que extrae la función que se encuentra en el primer `Just` y la mapea sobre`Just "volta"`, lo cual devuelve `Just "johntravolta"`. Si alguno de los dos valores hubiera sido `Nothing`, el resultado habría sido `Nothing`.

Hasta ahora, solo hemos usado `Maybe` en nuestros ejemplos y puede que estés pensado que los funtores aplicativos solo funcionan con `Maybe`. Existen un buen puñado de instancias de `Applicative`, así que vamos a probarlas.

Las listas (en realidad, el constructor de tipos `[]`) son funtores aplicativos ¡Qué sorpresa! Aquí tienes la instancia de `[]` para`Applicative`:

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

Antes dijimos que `pure` toma un valor y lo inserta en un contexto por defecto. En otras palabras, un contexto mínimo que contenga ese valor. El contexto mínimo para las listas sería la lista vacía, `[]`, pero la lista vacía representa el hecho de tener un valor, así que no puede mantener un valor por si mismo. Por este motivo, `pure` toma un valor y lo introduce en una lista unitaria. De forma similar, el contexto mínimo para el funtor aplicativo de `Maybe` sería `Nothing`, pero este representa el hecho de no tener un valor, así que `pure` está implementado usando `Just`.

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

¿Qué pasa con `<*>`? Si vemos el tipo de `<*>` como si estuviera limitado a las listas tendríamos algo como `(<*>) :: [a -> b] -> [a] -> [b]`. Está implementado usado *listas por comprensión*. `<*>` debe extraer de alguna forma la función que contiene el primer parámetro y mapearla sobre el segundo parámetro. El problema es que aquí puede haber una función, varias de ellas, o incluso ninguna. La lista de la derecha también puede contener varios valores. Por este motivo se utiliza una lista por comprensión para extraer valores de ambas listas. Aplicamos cada posible función de la lista de la izquierda en cada posible valor de la lista de la derecha. El resultado será una lista con cada posible combinación de aplicar una función de la primera lista sobre un valor de la segunda lista.

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

La lista de la izquierda tiene tres funciones y la lista de la derecha tiene tres valores, así que el resultado tendrá nueve elementos. Cada función de la lista de la izquierda se aplica a cada valor de la lista de la derecha. Si tuviéramos funciones que tomen dos parámetros, podemos aplicar estas funciones entre dos listas.

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

Como `<*>` es asociativo por la izquierda, lo primero que se resuelve es`[(+),(*)] <*> [1,2]`, que da como resultado una lista como esta`[(1+),(2+),(1*),(2*)]`, ya que cada función de la lista de la izquierda se aplica a cada valor de la lista de la derecha. Luego, se calcula`[(1+),(2+),(1*),(2*)] <*> [3,4]`, que devuelve el resultado anterior.

Usar el estilo aplicativo con listas es divertido. Mira:

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

De nuevo, fíjate en que hemos usado una función normal que toma dos cadenas entre dos funtores aplicativos de cadenas simplemente insertando los operadores aplicativos apropiados.

Puedes ver las listas como computaciones no deterministas. Un valor como`100` o `"que"` puede ser visto como una computación determinista que solo tienen un valor, mientras que una lista como `[1,2,3]` puede ser visto como un computación

que no puede decidir que resultado queremos, así que nos muestra una lista con todos los resultados posibles. Así que cuando hacemos algo como `(+) <$> [1,2,3] <*> [4,5,6]`, puedes pensar que se trata de sumar dos computaciones no deterministas con `+`, para que produzca otra computación no determinista que esté incluso menos segura de que valor es el resultado final.

El estilo aplicativo con listas suele ser un buen remplazo par la listas por comprensión. En el segundo capítulo, queríamos saber todos los posibles productos entre `[2,5,10]` y `[8,10,11]`, así que hicimos esto:

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

Simplemente extraemos valores de las dos listas y aplicamos una función para combinar los elementos. Esto también se puede hacer usando el estilo aplicativo:

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

En mi opinión la segunda versión es más clara, ya que es más fácil de ver que simplemente estamos aplicando `*` entre dos computaciones no deterministas. Si quisiéramos todos los posibles productos entre ambas listas que fueran mayores que 50, podríamos hacer algo como:

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

Es fácil de ver como `pure f <*> xs` es igual a `fmap f xs` con la listas.`pure f` es `[f]` y `[f] <*> xs` aplicará cada función que esté en la primera lista sobre cada valor que este en la segunda lista, pero solo hay una función en la lista de la izquierda, así que es como un `fmap`.

Otra instancia de `Applicative` con la que ya nos hemos encontrado es`IO`. Así es como se implementa:

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

Como todo lo que hace `pure` es insertar un valor en un contexto mínimo que pueda albergar ese valor, tiene sentido que `pure` sea simplemente`return`, ya que `return` hace exactamente eso: crea una acción de E/S que no hace nada, simplemente tiene como resultado el valor que le pasemos, pero en realidad no ejecuta ninguna operación de E/S como mostrar texto por un terminal o leer algo de algún fichero.

Si `<*>` fuera exclusivo para `IO` su tipo sería `(<*>) :: IO (a -> b) -> IO a -> IO b`. Tomaría una acción de E/S que devuelve una función como resultado y otra acción de E/S y crearía una una nueva acción de E/S a partir de estas dos, que cuando fuera ejecutada, primero ejecutaría la primera acción para obtener la función y luego ejecutaría la segunda acción para obtener un valor que luego aplicaría a la primera función para obtener el resultado de la acción que crea. Hemos utilizado la *sintaxis do* para implementarlo. Recuerda que la *sintaxis do* trata de tomar varias acciones de E/S y unirlas en una sola, que es exactamente lo que hacemos aquí.

Con `Maybe` y `[]`, podemos que pensar que `<*>` simplemente extrae una función de su parámetro izquierdo y luego lo aplica al de la derecha. Con`IO`, seguimos extrayendo una función, pero ahora también existe una*secuenciación* , ya que estamos tomando dos acciones de E/S y las estamos secuenciando, o uniéndolas, en una sola acción. Hay que extraer una función de la primera acción de E/S, pero para extraer un resultado de una acción de E/S, primero tiene que ser ejecutada.

Considera esto:

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

Esta acción de E/S preguntará al usuario por dos líneas de texto y las devolverá concatenadas. Esto se consigue gracias a que hemos unido dos acciones de E/S `getLine` y un `return`, ya que queríamos una nueva acción de E/S que contuviera el resultado `a ++ b++`. Otra forma de escribir esto sería usando el estilo aplicativo.

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

Lo que hacíamos antes era crear una acción de E/S que aplicará una función entre los resultados de otras dos acciones de E/S, y esto es exactamente lo mismo. Recuerda, `getLine` es una acción de E/S con el tipo`getLine :: IO String`. Cuando utilizamos `<*>` entre dos funtores aplicativos, el resultado es un funtor aplicativo, así que parece que tiene sentido.

Si volvemos a la analogía de la caja, podemos imaginar a `getLine` como una caja que viajará al mundo real y nos traerá una cadena. Al hacer`(++) <$> getLine <*> getLine` creamos un nueva caja más grande, que enviará esas dos cajas para obtener las dos líneas de la terminal y devolver la concatenación de ambas como resultado.

El tipo de la expresión `(++) <$> getLine <*> getLine` es `IO String`, esto quiere decir que esta expresión es una acción de E/S normal y corriente que también contiene un resultado, al igual que todas las demás acciones de E/S. Por esta razón podemos hacer cosas como esta:

```
main = do
    a <- (++) <$> getLine <*> getLine
    putStrLn $ "Las dos líneas concatenadas son: " ++ a
```

Si alguna vez te encuentras ligando una acción de E/S a algún nombre y luego utilizas una función sobre ella para luego devolver ese valor como resultado usando `return`, considera utilizar el estilo aplicativo ya que es sin duda alguna más conciso.

Otra instancia de `Applicative` es `(-> r)`, es decir, funciones. No es una instancia muy utilizada, pero sigue siendo interesante como aplicativo, así que vamos a ver como se implementa.

Nota

Si estas confudido acerca del significado de `(-> r)`, revisa la sección anterior donde explicamos como `(-> r)` es un funtor.

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

Insertamos un valor dentro de un funtor aplicativo con `pure`, el resultado que devuelva éste siempre debe ser el valor anterior. El contexto mínimo que siga conteniendo ese valor como resultado. Por este motivo en la implementación de la instancia funtor de las funciones, `pure` toma un valor y crea una función que ignora su parámetro y devuelve siempre ese mismo valor. Si vemos el tipo de `pure`, pero restringido al tipo de la instancia`(-> r)`, sería `pure :: a -> (r -> a)`.

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

Gracias a la currificación, la aplicación de funciones es asociativa por la izquierda, así que podemos omitir los paréntesis.

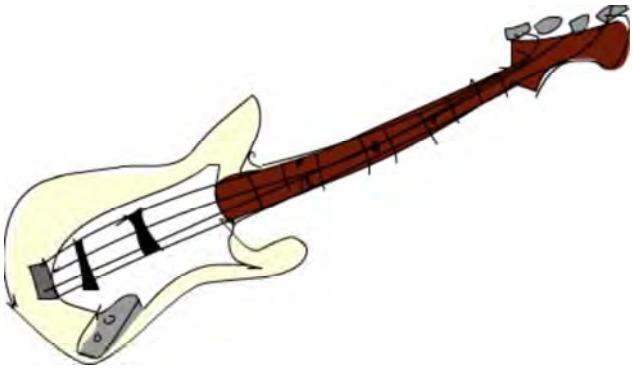
```
ghci> pure 3 "blah"
3
```

La implementación de la instancia para `<*>` es un poco críptica, así que será mejor si simplemente vemos un ejemplo de como utilizar las funciones como funtores aplicativos en estilo aplicativo:

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

Al llamar `<*>` con dos funtores aplicativos obtenemos otro funtor aplicativo, así que si utilizamos dos funciones, obtenemos de nuevo una función. Así que, ¿Qué sucede aquí? Cuando hacemos`(+) <$> (+3) <*> (*100)`, creamos una función que utilizará `+` en los resultados de `(+3)` y `(*100)` y devolverá ese resultado. Para demostrar este ejemplo real, hemos hecho `(+) <$> (+3) <*> (*100) $ 5`, el primer`5` se aplica a `(+3)` y `(*100)`, obteniendo `8` y `500`. Luego, se llama a `+` con `8` y `500`, obteniendo `508`.

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



Lo mismo. Hemos creado una función que llamará a `\x y z -> [x,y,z]` con los resultados finales de `(+3)`, `(*2)` y `(/2)`. El `5` será pasado a estas tres funciones y luego se llamará a `\x y z -> [x, y, z]` con los resultados.

Puedes verlo como si las funciones fueran cajas que contienen los resultados finales, así que si hacemos `k <$> f <*> g` se crea una función que llamará a `k` con los resultados de `f` y `g`. Cuando hacemos algo como`(+) <$> Just 3 <*> Just 5`, estamos usando `+` en valores que pueden estar ahí o no, por lo tanto el resultado será un valor o ninguno. Cuando hacemos algo como `(+) <$> (+10) <*> (+5)`, estamos usando `+` en los futuros resultados de las funciones `(+10)` y `(+5)`, y el resultado también será algo que producirá un valor siempre y cuando sea llamado con un parámetro.

No solemos utilizar las funciones como funtores aplicativos, pero siguen siendo interesantes. Tampoco es muy importante que no entiendas como funciona la instancia de las funciones para los funtores aplicativos, así que no te preocupes mucho. Intenta jugar un poco con el estilo aplicativo y las funciones para hacerte una idea de como funionan.

Una instancia de `Applicative` que aún no nos hemos encontrado es`ZipList` y reside en `Control.Applicative`.

Este tipo sugiere que en realidad hay mas formas de utilizar las listas como funtores aplicativos. Una forma es la que ya hemos visto, cuando utilizamos`<*>` con una lista de funciones y una lista de valores devuelve una lista de todas las posibles combinaciones de aplicar esas funciones de la lista de la izquierda a los valores de la derecha. Si hacemos algo como`[(+3), (*2)] <*> [1,2]`, `(+3)` será aplicado a `1` y `2` y `(*2)`también será aplicado a ambos, por lo que obtendremos una lista con cuatro elementos, `[4,5,2,4]`.

Sin embargo, `[(+3),(*2)] <*> [1,2]` también podría funcionar de forma que la primera función de la izquierda fuera aplicada a el primer valor de la derecha y la segunda función fuera aplicada al segundo valor. Esto nos daría una lista con dos valores, `[4,4]`. Lo podríamos ver como`[1 + 3, 2 * 2]`.

Como un mismo tipo no puede tener dos instancias para una misma clase de tipos, se utiliza el tipo `ZipList a`, que tiene un constructor `ZipList` con un solo campo, la lista. Aquí esta la instancia:

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

Nota
Sí, también sería válido `ZipList (zipWith ($) fs xs)`.

`<*>` hace lo que acabamos de explicar. Aplica la primera función a el primer valor, la segunda función al segundo valor, etc. Esto se consigue con`zipWith (\f x -> f x) fs xs`. Debido a como funciona `zipWith`, la lista final será tan larga como la lista más corta de las dos.

`pure` es bastante interesante. Toma un valor y lo introduce en una lista que tiene ese valor repetido indefinidamente. `pure "jaja"` devolvería algo como `ZipList (["jaja","jaja","jaja"...`. Quizá esto sea algo confuso ya que hemos dicho que `pure` debe introducir un valor en el contexto mínimo que albergue ese valor. Y quizá estés pensado que una lista infinita difícilmente es un contexto mínimo. Pero tiene sentido con esta listas, ya que tiene que producir un valor en cada posición. De esta forma también se cumple la ley que dice que `pure f <*> xs` debe ser igual a `fmap f xs`. Si `pure 3` solo devolviera `ZipList [3]`, `pure (*2) <*> ZipList [1,5,10]` devolvería`ZipList [2]`, ya que la lista resultante es tan larga como la mas corta de las dos que utilizamos como parámetros. Si utilizamos una lista infinita y otra finita, la lista resultante siempre tendrá el tamaño de la lista finita.

¿Cómo funcionan estas listas al estilo aplicativo? Veamos. El tipo`ZipList a` no tiene una instancia para `Show`, así que tenemos que utilizar la función `getZipList` para extraer una lista primitiva.

```
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 lo mismo que `\x y z -> (x,y,z)`. También, la función `(,)` sería igual a `\x y -> (x,y)`.

A parte de `zipWith`, la biblioteca estándar también tiene funciones como`zipWith3`, `zipWith4`, y todas las demás hasta llegar a 7. `zipWith`toma una función que tome dos parámetros y une dos los listas con esta función. `zipWith3` toma una función que tome tres parámetros y une tres listas con ella. Gracias a estas listas y al estilo aplicativo, no tenemos que tener una función

distinta para cada número de listas que queramos unir con una función. Lo único que tenemos que hacer es utilizar el estilo aplicativo.

`Control.Applicative` define una función llamada `liftA2`, cuyo tipo es `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
```

Nada especial, simplemente aplica una función entre dos funtores aplicativos, escondiendo el estilo aplicativo al que nos hemos acostumbrado. La razón por la cual lo mostramos es para hacer más evidente porque los funtores aplicativos son más potentes que los funtores ordinarios. Con lo funtores ordinarios solo podemos mapear funciones sobre un funtor. Pero con los funtores aplicativos, podemos aplicar una función con varios funtores. También es interesante ver la el tipo de la función como`(a -> b -> c) -> (f a -> f b -> f c)`. Si lo vemos de esta forma, podemos decir que `liftA2` toma una función binaria normal y la desplaza para que opere con dos funtores.

Un concepto interesante: podemos tomar dos funtores aplicativos y combinarlos en un único funtor aplicativo que contenga los resultados de ambos funtores en forma de lista. Por ejemplo, tenemos `Just 3` y `Just 4`. Vamos a asumir que el segundo está dentro de una lista unitaria, lo cual es realmente fácil de conseguir:

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

Vale, ahora tenemos `Just 3` y `Just [4]` ¿Cómo obtendríamos`Just [3,4]`? Fácil.

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

Recuerda, `:` es una función que toma un elemento y una lista y devuelve una lista nueva con dicho elemento al principio. Ahora que tenemos `Just [3,4]`, ¿podríamos combinarlos con `Just 2` para obtener `Just [2,3,4]`? Por supuesto que podríamos. Parece que podemos combinar cualquier cantidad de funtores aplicativos en uno que contenga una lista con los resultados de dichos funtores. Vamos a intentar implementar una función que tome una lista de funtores aplicativos y devuelva un funtor aplicativo que contenga una lista con los resultados de los funtores. La llamaremos `sequenceA`.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

¡Ahh, recursión! Primero, veamos su tipo. Transformará una lista funtores aplicativos en un funtor aplicativo con un lista. Esto nos da alguna pista para el caso base. Si queremos convertir una lista vacía en un funtor aplicativo con una lista que contenga los resultados, simplemente insertamos la lista en el contexto mínimo. Luego viene la recursión. Si tenemos una lista con una cabeza y una cola (recuerda, `x` es un funtor aplicativo y `xs` es una lista de ellos), llamamos a `sequenceA` con la cola para que nos devuelva un funtor aplicativo que contenga una lista. Luego, anteponemos el valor que contiene el funtor aplicativo `x` en la lista ¡Y listo!

Si hiciéramos `sequenceA [Just 1, Just 2]`, tendríamos`(:) <$> Just 1 <*> sequenceA [Just 2]`, que es igual a `(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])`. Sabemos que`sequenceA []` acabará siendo `Just []`, así que ahora tendríamos`(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])`, que es igual a`(:) <$> Just 1 <*> Just [2]`, que es igual a `Just [1,2]`.

Otra forma de implementar `sequenceA` es con un pliegue. Recuerda, casi cualquier función en la que recorramos una lista elemento a elemento y vayamos acumulando un resultando a lo largo del camino puede ser implementada con un pliegue.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

Empezamos recorriendo la lista por la izquierda y con un acumulador inicial igual a `pure []`. Aplicamos `liftA2 (:)` entre el acumulador y el último elemento de la lista, lo cual resulta en un funtor aplicativo que contiene una lista unitaria. Luego volvemos a aplicar `liftA2 (:)` con el último elemento actual de la lista con el acumulador actual, y así sucesivamente hasta que solo nos quedemos con el acumulador, que contendrá todos los resultados de los funtores aplicativos.

Vamos a probar nuestra función.

```
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],[]]
[]
```

Precioso. Cuando lo utilizamos con los valores `Maybe`, `sequenceA` crea un valor `Maybe` con todos los resultados dentro de una lista. Si alguno de los valores es `Nothing`, entonces el resultado final también lo es. Esto puede ser útil cuando tenemos una lista de valores `Maybe` y estamos interesados en obtener esos valores solo si ninguno de ellos es `Nothing`.

Cuando se utiliza con funciones, `sequenceA` toma una lista de funciones y devuelve una función cuyo resultado es una lista. En el ejemplo anterior, creamos una función que tomará un número como parámetro, se aplica a cada una de las funciones de la lista y luego devuelve una lista con los resultados.`sequenceA [(+3),(+2),(+1)] 3` llamará a `(+3)` con `3`, a `(+2)` con`3` y a `(+1)` con `3`, luego devolverá una lista con todos los resultados.

Si hacemos `(+) <$> (+3) <*> (*2)` estamos creando una función que toma un parámetro, lo aplica a `(+3)` y a `(*2)` y luego llama a `+` con ambos resultados. Del mismo modo, si hacemos `sequenceA [(+3),(*2)]` estamos creando una función que tomará un parámetro y lo aplicará a las funciones de la lista. Pero, en lugar de llamar a `+` con los resultados de las funciones, se utiliza una combinación de `:` y `pure []` para unir todos esos resultados en una lista.

`sequenceA` puede ser útil cuando tenemos una lista de funciones y queremos aplicarlas todas al mismo parámetro y luego tener los resultados en una lista. Por ejemplo, si tenemos un número y queremos saber si satisface todos los predicados que contiene una lista. Una forma de hacerlo sería así:

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

Recuerda que `and` toma una lista de booleanos y devuelve `True` si son todos `True`. Otra forma de hacer lo mismo sería 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 tomará un número y lo aplicará a todos los predicados de la lista, `[(>4),(<10),odd]`, y devolverá una lista con los resultados. Dicho de otra forma, convierte una lista de tipo`(Num a) => [a -> Bool]` en una función cuyo tipo sería`(Num a) => a -> [Bool]` ¿Tiene buena pinta, no?

Ya que las listas son homogéneas, todas las funciones de la lista deben tener el mismo tipo. No podemos tener una lista como `[ord, (+3)]`, porque `ord`toma un carácter y devuelve un número, mientras que `(+3)` toma un número y devuelve otro número.

Cuando se utiliza con `[]`, `sequenceA` toma una lista y devuelve una lista de listas. Mmm... interesante. En realidad crea una lista que contiene todas las combinaciones posibles de sus elementos. A título de ejemplo aquí tienes unos cuantos usos de `sequenceA` con sus equivalentes usando listas por comprensión:

```
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> [[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]]
```

Quizá esto es un poco difícil de entender, pero si jugamos un poco con ellos, veremos como funciona. Digamos que tenemos `sequenceA [[1,2],[3,4]]`. Para ver lo que sucede, vamos a utilizar la definición`sequenceA (x:xs) = (:) <$> x <*> sequenceA xs` de `sequenceA` y el caso base `sequenceA [] = pure []`. No tienes porque seguir esta traza, pero si no consigues imaginarte como funciona `sequenceA` con las listas puede que te resulte de ayuda.

- Empezamos con `sequenceA [[1,2],[3,4]]`.
- Lo cual se evalúa a `(:) <$> [1,2] <*> sequenceA [[3,4]]`.
- Si evaluamos el `sequenceA` interior una vez más, obtenemos`(:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])`.
- Ahora hemos alcanzado el caso base, así que tenemos`(:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])`.
- Evaluamos la parte `(:) <$> [3,4] <*> [[]]`, que utilizará `:` con cada posible valor de la lista de la izquierda (es decir `3` y `4`) con cada posible valor de la lista de la derecha (`[]`), obteniendo así `[3:[], 4:[]]`, que es `[[3],[4]]`. Así que ahora tenemos`(:) <$> [1,2] <*> [[3],[4]]`.
- `:` se utiliza con cada posible valor de la lista de la izquierda (`1` y `2`) con cada posible valor de la lista de la derecha (`[3]` y`[4]`), de forma que nos quedamos con `[1:[3], 1:[4], 2:[3], 2:[4]]`, que es igual a `[[1,3],[1,4], [2,3],[2,4]`.

Si hacemos `(+) <$> [1,2] <*> [4,5,6]` estamos creando una computación no determinista `x + y` donde `x` toma cualquier valor de `[1,2]` e`y` toma cualquier valor de `[4,5,6]`. Representamos la solución con una lista con todos los posibles resultados. De forma similar, si hacemos`sequence [[1,2],[3,4],[5,6],[7,8]]` estamos creando una computación no determinista `[x,y,z,w]`, donde `x` toma cualquier valor de `[1,2]`,`y` toma cualquier valor de `[3,4]`, y así sucesivamente. Representamos el resultado de la computación no determinista utilizando una lista, donde cada elemento es una lista posible. Por este motivo el resultado es una lista de listas.

Con acciones de E/S, `sequenceA` se comporta igual que `sequence`. Toma una lista de acciones de E/S y devuelve una acción de E/S que ejecutará cada una de esas acciones y tendrá como resultado una lista con los resultados de todas esas acciones. Por este motivo para convertir un valor `[IO a]` en un valor `IO [a]`, o dicho de otra forma, para crear una acción de

E/S que devuelva una lista de resultados cuando sea ejecutada, todas estas acciones tienen que ser secuenciadas de forma que sean ejecutadas unas detrás de otra cuando se fuerce la evaluación. No puede obtener el resultado de una acción de E/S si no la ejecutas primero.

```
ghci> sequenceA [getLine, getLine, getLine]
heyh
ho
woo
["heyh","ho","woo"]
```

Al igual que los funtores normales, los funtores aplicativos vienen con una serie de leyes. La mas importante de todas es la que ya hemos mencionado,`pure f <*> x = fmap f x`. Como ejercicio, puedes intentar comprobar esta ley en algunos de los funtores de los que hemos hablado. Las otras leyes son:

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

No vamos a verlas en detalle ahora mismo ya que nos tomaría unas cuantas página y probablemente sea algo aburrido, pero, si te sientes con ganas, puedes echarles una vistazo más de cerca y comprobar si algunas de las instancias que hemos visto las cumplen.

Concluyendo, los funtores aplicativos no son solo interesantes, sino que también son útiles, ya que nos permiten combinar diferentes computaciones, como computaciones de E/S, computaciones no deterministas, computaciones que puede fallar, etc. utilizando el estilo aplicativo. Simplemente utilizando`<$>` y `<*>` podemos utilizar funciones normales para que operen de forma uniforme con cualquier número de funtores aplicativos y tomar ventaja de la semántica de cada uno.

La palabra clave newtype

Hasta ahora hemos creado nuestros propios tipos de datos algebraicos utilizando el palabra clave `data`. También hemos visto como dar sinónimos de tipos ya existentes utilizando la palabra clave `type`. En esta sección, veremos como crear nuevos tipos de datos a partir de tipos datos ya existentes utilizando la palabra clave `newtype` y el porqué de hacerlo de este modo.

En la sección anterior vimos que en realidad hay más de una forma para que una lista sea un funtor aplicativo. Una manera es que `<*>` tome cada función de la lista que se le pasa como parámetro izquierdo y la aplique a cada valor que contenga la lista de la derecha, de forma que devuelva todas las posibles combinaciones de aplicar una función de la izquierda con un valor de la derecha.

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

La segundo forma es que tome la primera función de la lista de la izquierda de`<*>` y la aplique a el primer valor de la lista de la derecha, luego tomará la segunda función de la lista izquierda y la aplicara al segundo valor de la lista derecha, y así sucesivamente. Al final es algo como unir dos listas en una. Pero las listas ya tienen una instancia para `Applicative`, así que ¿Cómo hemos creado una segunda instancia de `Applicative`? Si haces memoria, recordarás que dijimos que el tipo `ZipList a` se utilizaba por este motivo, el cual tiene un constructor de datos, `ZipList`, con un solo campo. Pusimos la lista con la que íbamos a trabajar en ese campo. Luego, como `ZipList`tenia su propia instancia de `Applicative`, el comportamiento de las listas como funtores aplicativos era diferente. Solo teníamos que utilizar el constructor `ZipList` con la lista y cuando termináramos debíamos usar`getZipList` para recuperarla.

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3]
[2,200,15]
```

Y bien ¿Qué tiene que ver todo esto con la palabra clave `newtype`? Bueno, piensa un poco en como deberíamos declarar el tipo de datos `ZipList a`. Una forma sería así:

```
data ZipList a = ZipList [a]
```

Un tipo que solo tiene un constructor de datos y este constructor solo tiene un campo el cual es una lista de cosas. También podríamos utiliza la sintaxis de registro para obtener de forma automática una función que extraiga la lista de un `ZipList`:

```
data ZipList a = ZipList { getZipList :: [a] }
```

Todo esto parece correcto y de hecho funciona bien. Simplemente hemos utilizado la palabra clave `data` para insertar un tipo dentro dentro de otro y así poder crear una segunda instancia de el tipo original.

En Haskell, la palabra clave `newtype` se utiliza exactamente para estos casos en los que simplemente queremos insertar un tipo dentro de otro para que parezca un tipo distinto. En realidad, `ZipList` se define así:

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

Se utiliza `newtype` en lugar de `data`. Y ¿Por qué? Te estarás preguntando. Muy sencillo, `newtype` es más rápido. Si utilizamos la palabra clave `data` para insertar un tipo dentro de otro, se genera cierta sobrecarga cuando el programa se ejecuta debido a las operaciones que insertan y extraen el tipo. Pero si utilizamos `newtype`, Haskell sabe que lo estamos utilizando para insertar un tipo existente en un nuevo tipo (de ahí viene el nombre). En realidad lo que buscamos es que internamente sean iguales pero que su tipo sea distinto. Teniendo esto en cuenta, Haskell puede deshacerse de las operaciones de inserción y extracción una vez sepa de que tipo es cada valor.

Entonces ¿Por qué no utilizamos siempre `newtype` en lugar de `data`? Cuando creamos un nuevo tipo a partir de uno ya existente utilizando la palabra clave `newtype`, solo podemos utilizar un constructor de datos y éste solo puede tener un campo. Mientras que con `data` podemos tener varios constructores de datos y cada uno de ellos con cero o varios campos.

```
data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession
```

Cuando utilizamos `newtype` estamos restringidos a utilizar a utilizar un solo constructor con un solo campo.

También podemos utilizar la palabra clave `deriving` con `newtype` de la misma forma que hacemos con `data`. Podemos derivar las instancias de`Eq`, `Ord`, `Enum`, `Bounded`, `Show` y `Read`. Si derivamos la instancia de una clase de tipos, el tipo original tiene que ser miembro de dicha clase de tipos. Tiene sentido, ya que `newtype` solo sustituye a un tipo existente. Si tenemos el siguiente código, podríamos mostrar por pantalla y equiparar valores del nuevo tipo:

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

Vamos a probarlo:

```
ghci> CharList "this will be shown!"
CharList {getCharList = "this will be shown!"}
ghci> CharList "benny" == CharList "benny"
True
ghci> CharList "benny" == CharList "oisters"
False
```

En este caso en particular, el constructor de datos tiene el siguiente tipo:

```
CharList :: [Char] -> CharList
```

Toma un valor del tipo `[Char]`, como `"My Sharona"` y devuelve un valor del tipo `CharList`. En el ejemplo anterior lo podemos ver en funcionamiento. Po el contrario, la función `getCharList`, que ha sido generada automáticamente gracias al uso de la sintaxis de registro, tiene este tipo:

```
getCharList :: CharList -> [Char]
```

Toma un valor del tipo `CharList` y devuelve uno del tipo `[Char]`. Estas son las operaciones de inserción y extracción de las que antes hablábamos, aunque también puedes verlo como una transformación de un tipo a otro. Gracias a las propiedades de `newtype`, estas operaciones no tendrán ningún coste en tiempo de ejecución.

Utilizando newtype para crear instancias de clase

A menudo queremos crear instancias de nuestros tipos para ciertas clases de tipos, pero los parámetros de tipo no encajan en lo que queremos hacer. Es muy fácil crear una instancia de `Maybe` para `Functor`, ya que la clase de tipos `Functor` se define como:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Así que simplemente tenemos que hacer esto:

```
instance Functor Maybe where
```

E implementar `fmap`. Todos los parámetros de tipo encajan porque `Maybe`toma el lugar de `f` en la definición de la clase de tipos `Functor`, de forma que si vemos el tipo de `fmap` como si solo funcionara para `Maybe`quedaría así:

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

Ahora ¿Qué pasaría si quisiéramos crear una instancia para `Functor` para las duplas de forma que cuando utilizamos `fmap` con una función sobre una dupla, se aplicara la función al primer componente de la dupla? De este modo, si hiciéramos algo como `fmap (+3) (1,1)` obtendríamos `(4,1)`. Pues resulta que escribir una instancia para lograr este comportamiento no es tan sencillo. Con `Maybe` solo teníamos que utilizar`instance Functor Maybe where` porque solo los constructores de tipos que toman exactamente un parámetro pueden crear una instancia para la clase`Functor`. Pero parece que no hay ninguna forma de hacer que algo como que el parámetro `a` de `(a,b)` acabe siendo el que cambie cuando utilicemos`fmap`. Para solucionarlo, podemos utilizar `newtype` con las duplas de forma que el segundo parámetro de tipo represente el primer parámetro de tipo de las duplas:

```
newtype Pair b a = Pair { getPair :: (a,b) }
```

Y ahora podemos hacer la instancia para `Functor` de forma que la función sea aplicada únicamente en la primera componente:

```
instance Functor (Pair c) where
    fmap f (Pair (x,y)) = Pair (f x, y)
```

Como puede observar, podemos utilizar el ajuste de patrones con tipos definidos con `newtype`. Utilizamos el ajuste de patrones para obtener la dupla subyacente, luego aplicamos la función `f` al primer componente de la tupla y luego utilizamos el constructor de datos `Pair` para convertir la tupla de nuevo al tipo `Pair b a`. El tipo de `fmap` restringido al nuevo tipo quedaría así:

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

De nuevo, hemos utilizado `instance Functor (Pair c) where` así que`(Pair c)` toma el lugar de `f` en la definición de clase de `Functor`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Ahora podemos convertir una dupla en un `Pair b a`, y utilizar `fmap`sobre ella de forma que la función solo se aplique a la primera componente:

```
ghci> getPair $ fmap (*100) (Pair (2,3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("london calling", 3))
("gnillac nodnol",3)
```

La pereza de newtype

Ya hemos hablado de que normalmente `newtype` es más rápido que `data`. Lo único que podemos hacer cuando utilizamos `newtype` es convertir un tipo existente en un nuevo tipo, aunque internamente, Haskell puede representar los valores de los tipos definidos con `newtype` igual que los originales, aunque debe tener en cuenta que sus tipos son diferentes. Esto provoca que`newtype` no solo sea más rápido, sino también más perezoso. Vamos a ver con detalle que significa esto.

Como ya dijimos, Haskell perezoso por defecto, lo que significa que solo cuando intentamos mostrar el resultado de nuestras funciones tendrá lugar el cómputo de estos resultados. Además, solo los cómputos que son necesario para calcular el resultado la función serán ejecutados. El valor `undefined` de Haskell representa un cómputo erróneo. Si intentamos evaluarlo (es decir, forzamos a Haskell a que lo calcule) mostrándolo por la terminal, Haskell se lanzará un berrinche (técnicamente conocido como excepción):

```
ghci> undefined
*** Exception: Prelude.undefined
```

Sin embargo, si insertamos algunos valores `undefined` en una lista pero solo necesitamos la cabeza de la lista, la cual no es `undefined`, todo funcionará bien ya que Haskell no necesita evaluar ningún otro elemento de la lista si solo estamos interesados en el primer elemento:

```
ghci> head [3,4,5,undefined,2,undefined]
3
```

Ahora consideremos el siguiente tipo:

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

Es uno de los muchos tipos de datos algebraicos que se pueden definir con la palabra clave `data`. Tiene un único constructor de datos, y este constructor solo posee un campo cuyo tipo es `Bool`. Vamos a crear una función que use un ajuste de patrones en un `CoolBool` y devuelva el valor `"hola"` independientemente de que el valor `Bool` contenido en `CoolBool` sea `True` o `False`:

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hola"
```

En lugar de aplicar esta función a un valor normal de `CoolBool`, vamos a complicarnos la vida y aplicar el valor `undefined`.

```
ghci> helloMe undefined
"*** Exception: Prelude.undefined
```

¡Una excepción! ¿Por qué sucede esto? Los tipos definidos con la palabra clave `data` pueden tener varios constructores de datos (aunque `CoolBool` solo tiene uno). Así que para saber si un valor dado a nuestra función se ajusta al patrón `(CoolBool _)`, Haskell tiene que evaluar el valor lo suficiente como para saber el constructor de datos que se ha utilizado para crear el valor. Y cuando tratamos de evaluar un valor `undefined`, por muy poco que lo evaluemos, se lanzará una excepción.

En lugar de utilizar la palabra clave `data` para definir `CoolBool`, vamos a intentar utilizar `newtype`:

```
newtype CoolBool = CoolBool { getCoolBool :: Bool }
```

No tenemos que cambiar nada de la función `helloMe` porque la sintaxis que se utiliza en el ajuste de patrones es igual para `data` que para `newtype`. Vamos a hacer lo mismo y aplicar `helloMe` a un valor `undefined`:

```
ghci> helloMe undefined
"hola"
```

¡Funcionó! Mmm... ¿Por qué? Bueno, como ya hemos dicho, cuando utilizamos `newtype`, Haskell puede representar internamente los valores del nuevo tipo como si se tratasen del original. No tiene que añadir ningún envoltorio a estos valores, simplemente debe tener en cuenta de que poseen un tipo distinto. Y como Haskell sabe que los tipos definidos con la palabra clave `newtype` solo pueden tener un constructor de datos, no tiene porque evaluar el parámetro pasado a la función para estar seguro de que se ajusta al patrón `(CoolBool _)` ya que los tipos `newtype` solo pueden tener un constructor de datos con un solo campo.

Esta diferencia de comportamiento puede parecer trivial, pero en realidad es muy importante ya que nos ayuda a entender que aunque los tipos definidos con `data` y `newtype` se comportan de forma muy similar desde el punto de vista de un programador, en realidad son dos mecanismos diferentes. Mientras `data` se puede utilizar para crear nuestros propios tipos de datos desde cero, `newtype` sirve para crear un tipo completamente nuevo a partir de uno ya existente. Cuando utilizamos un ajuste de patrones con un tipo `newtype` no estamos extrayendo ningún dato de él (como ocurriría con `data`), sería más bien como una conversión directa entre un dato y otro.

type vs. newtpe vs. data

Llegados a este punto, quizás estés algo confundido sobre que diferencias existen entre `type`, `data` y `newtype`. Vamos a refrescar la memoria.

La palabra clave `type` se utiliza para crear sinónimos. Básicamente lo que hacemos es dar otro nombre a un tipo que ya existe de forma que nos sea más fácil referirnos a él. Por ejemplo:

```
type IntList = [Int]
```

Todo lo que hace es permitirnos llamar al tipo `[Int]` como `IntList`. Se puede utilizar indistintamente. No obtenemos ningún constructor de datos nuevo a partir de `IntList` ni nada por el estilo. Como `[Int]` y `IntList` son dos formas de referirse al mismo tipo, no importa que nombre usemos en las declaraciones de tipo:

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

Utilizamos los sinónimos de tipos cuando queremos que nuestras declaraciones de tipo sean más descriptivas, de forma que los sinónimos que demos expliquen algo acerca de su propósito en un determinado contexto. Por ejemplo, si utilizamos listas de asociación del tipo `[(String,String)]` para representar una agenda telefónica, podemos darle el sinónimo de tipo`PhoneBook` de forma que las declaraciones de tipo de las funciones sean más legibles.

La palabra clave `newtype` se utiliza para crear nuevos tipos a partir de uno ya existente. Su uso es común para facilitar la declaración de ciertas instancias de clases de tipos. Cuando utilizamos `newtype` con un tipo ya existente, el tipo que obtenemos es diferente del original. Si tenemos el siguiente tipo:

```
newtype CharList = CharList { getCharList :: [Char] }
```

No podemos utilizar `++` para concatenar un `CharList` con un `[Char]`. Ni siquiera podemos utilizar `++` para concatenar dos `CharList` porque`++` solo funciona con listas. `CharList` no es una lista, incluso aunque sepamos que contiene una. Sin embargo, podemos convertir dos `CharList` en listas, luego utilizar `++` con ellas y más tarde convertir el resultado en un `CharList`.

Cuando utilizamos la sintaxis de registro en las declaraciones `newtype`, obtenemos funciones para convertir ente el nuevo tipo y el original. El nuevo tipo no posee automáticamente las instancias para clases de tipos de las que formaba parte el tipo original, así que tenemos que derivarlas manualmente.

En la practica, puedes considerar las declaraciones de tipo `newtype`iguales a las declaraciones `data`, aunque solo puede tener un constructor de datos y un solo campo. Si te encuentras declarando un tipo como ese, plantéate utilizar `newtype` en lugar de `type`.

La palabra clave `data` la utilizamos para crear nuestros propios tipos de datos, y podemos hacer lo que se nos antoje con ellos. Pueden tener tantos constructores de datos y tantos campos como quieras y se pueden utilizar para implementar cualquier tipo de dato algebraico. Cualquier cosa, desde listas hasta tipos como `Maybe` o árboles.

Si quieres que las declaraciones de tipo sean más descriptivas y legibles, probablemente lo que estés buscando sean los sinónimos de tipos. Si lo que quieres es crear un nuevo tipo que contenga a otro para poder declarar una instancia de una clase de tipos, seguramente quieras utilizar `newtype`. Y si lo que quieres es crear algo completamente nuevo, apostaría a que debes utilizar `data`.

Monoides

En Haskell, las clases de tipos se utilizan crear una interfaz de un comportamiento que comparten

varios tipos. Empezamos viendo la sencillas clases de tipos, como `Eq`, que representa los tipos que pueden ser equiparados, o `Ord`, que representa los tipos cuyos valores pueden ser puestos en un determinado orden. Luego continuamos viendo clases de tipos más interesantes, como `Functor` o `Applicative`.

Cuando creamos un tipo debemos pensar en que comportamientos debe soportar, es decir, pensar si puede actuar como esos comportamientos, y luego decidir si es oportuno crear una instancia de estos. Si tiene sentido que los valores de un determinado tipo sean equiparados, entonces creamos una instancia de `Eq`para ese tipo. Si vemos que un tipo es un especie de funtor, podemos crear una instancia de `Functor`, y así sucesivamente.

Ahora consideremos esto: la función `*` toma dos números y los multiplica. Si multiplicamos un número por `1`, el resultado es siempre igual a ese número. No importa si hacemos `1 * x` o `x * 1`, el resultado es siempre el mismo. De forma similar, `++` también es una función que toma dos parámetros y devuelve una tercera. Solo que en lugar de multiplicar números, toma dos listas y las concatena. Al igual que pasaba con `*`, `++` también posee un valor que hará que el resultado final solo dependa del otro valor. En este caso el valor es la lista vacía, `[]`.

```
ghci> 4 * 1
4
ghci> 1 * 9
9
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [0.5, 2.5]
[0.5,2.5]
```

Parece que tanto `*` junto a `1` como `++` junto a `[]` comparten ciertas propiedades:

- La función toma dos parámetros.
- Los parámetros y el valor de retorno comparten el mismo tipo.
- Existe un valor que hará que el resultado de la función binaria solo dependa del otro parámetro.

Existe también otra propiedad que ambos comparte pero que quizá no sea tan obvia: cuando tenemos tres o mas valores y queremos utilizar la función binaria para reducirlos a un solo valor, el orden en el que apliquemos la función binaria no importa. No importa si hacemos `(3 * 4) * 5` o`3 * (4 * 5)`, al final el resultado será `60`. Lo mismo ocurre para`++`:

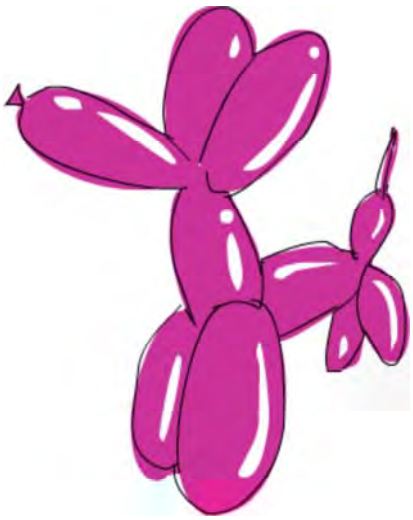
```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "la" ++ ("di" ++ "da")
"ladida"
ghci> ("la" ++ "di") ++ "da"
"ladida"
```

Llamamos a esta propiedad *asociatividad*. `*` es asociativa, y también lo es `++`, pero `-`, por ejemplo, no lo es. Las expresiones `(5 - 3) - 4`y `5 - (3 - 4)` tienen resultados diferentes.

Si observamos estas propiedades nos encontraremos con los *monoides*. Un monoide es cuando tienes una función binaria asociativa y valor que actúa como identidad respecto a esa función. Cuando se dice que un valor actúa como identidad respecto a una función significa que, cuando se utiliza en esa función junto con otro valor, el resultado siempre siempre es igual al otro valor. `1` es la identidad respecto a `*` y `[]` es la identidad respecto a `++`. En el mundo de Haskell existen muchos más monoides y por este motivo existe la clase de tipos `Monoid`. Es para tipos que pueden actuar como monoides. Vamos a ver como se define:

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

La clase `Monoid` está definida en `Data.Monoid`. Vamos a tomarnos un rato para familiarizarnos con ella.



Antes de nada, podemos ver que solo los tipo concretos pueden tener una instancia de `Monoid`, ya que `m`, en la definición de la clase, no toma ningún parámetro de tipo. Es diferente de lo que sucede con `Functor` y`Applicative`, ya que sus instancias requieren que los constructores de tipos tomen un parámetro.

La primera función es `mempty`. En realidad no es un función porque no toma ningún parámetro, así que es una constante polimórfica, parecido a`minBound` o `maxBound`. `mempty` representa el valor identidad para un determinado monoide.

A continuación tenemos `mappend`, que, como ya habrás adivinado, es la función binaría. Toma dos parámetros del mismo tipo y devuelve un valor del mismo tipo también. La decisión de llamar a esta función `mappend` (añadir) no fue la más correcta, ya que implica que de algún modo vamos añadir dos cosas. Mientras que `++` toma dos listas y añade una a la otra, `*` no añade nada, simplemente multiplica dos números. Cuando veamos más instancias de `Monoid`, veremos que muchas de ellas tampoco añaden valores con esta función, así que evita pensar en términos de añadir y piensa que `mappend`es la función binaria del monoide.

La última función definida por esta clase de tipos es `mconcat`. Toma una lista de monoides y la reduce y la reduce a uno solo valor aplicando`mappend` entre los elementos de la lista. Posee una implementación por defecto, que toma `mempty` como valor inicial y pliega la lista por la derecha con la función `mappend`. Como la implementación por defecto de`mconcat` es válida para la mayoría de la instancias, no nos vamos a preocupar mucho por ella. Cuando creamos una instancia de `Monoid` basta con implementar `mempty` y `mappend`. La razón por la que `mconcat` se encuentra en la declaración de clase es que para ciertas instancias, puede que exista una forma más eficiente de implementar `mconcat`, pero para la mayoría de la instancias la implementación por defecto es perfecta.

Antes de ver algunas de las instancias de `Monoid`, vamos a echar un pequeño vistazo a las leyes de los monoides. Hemos mencionado que debe existir un valor que actúe como identidad con respecto a la función binaria y que dicha función debe ser asociativa. Es posible crear instancias de `Monoid` que no sigan estas reglas, pero estas instancias no serán útiles para nadie ya que cuando utilizamos la clase `Monoid`, confiamos en que estas instancias se comporten como monoides De otro modo, ¿qué sentido tendría todo eso? Por esta razón, cuando creamos instancias debemos asegurarnos de cumplir estas leyes:

- `mempty `mappend` x = x`
- `x `mappend` mempty = x`
- `(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)`

Las primeras dos leyes establecen que `mempty` debe actuar como identidad respecto a `mappend` y la tercera dice que `mappend` debe ser asociativa, es decir, que el orden en el que utilicemos `mappend` para reducir varios valores de un monoide en

uno no debe importar. Haskell no comprueba estas leyes, así que nosotros como programadores debemos ser cautelosos y asegurarnos de obedecer estas leyes.

Las listas son monoides

¡Sí, las listas son monoides! Como ya hemos visto, la función `++` y la lista vacía `[]` forman un monoide. La instancia es muy simple:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Las listas poseen su propia instancia para la clase de tipos `Monoid` independientemente del tipo de dato que alberguen. Fíjate que hemos utilizado`instance Monoid [a]` y no `instance Monoid []`, ya que `Monoid`requiere un tipo concreto para formar la instancia.

Si realizamos algunas pruebas no nos encontraremos ninguna sorpresa:

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> ("one" `mappend` "two") `mappend` "tree"
"onetwotree"
ghci> "one" `mappend` ("two" `mappend` "tree")
"onetwotree"
ghci> "one" `mappend` "two" `mappend` "tree"
"onetwotree"
ghci> "pang" `mappend` mempty
"pang"
ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```

Fíjate en la última línea, hemos tenido que usar una anotación de tipo explícita, ya que si solo hubiésemos puesto `mempty`, `GHCi` no sabría que instancia usar así que tenemos que especificar que queremos utilizar la instancia de las listas. Hemos sido capaces de utilizar un tipo general como`[a]` (en lugar de especificar un tipo concreto como `[Int]` o`[String]`) porque las listas vacías puede actuar como si contuvieran cualquier tipo.

Como `mconcat` tiene una implementación por defecto, obtenemos esta función automáticamente cuando creamos una instancia de `Monoid`. En el caso de las listas`,` `mconcat` se comporta igual que `concat`. Toma una lista de listas y las une utilizando `++` entre entre las listas adyacentes contenidas en la lista.

Las leyes de los monoides se cumplen para la instancia de las listas. Cuando tenemos varias listas y utilizamos `mappend` (o `++`) para unirlas, no importa que unión hagamos primero, ya que al final acabaran siendo unidas de todas formas. También, la lista vacía actúa como identidad. Ten en cuenta que los monoides no requieren que `a `mappend` b` sea igual a `b `mappend` a` (es decir, no son conmutativos). En el caso de las listas, se puede observar fácilmente:

```
ghci> "one" `mappend` "two"
"onetwo"
ghci> "two" `mappend` "one"
"twoone"
```

No pasa nada. El hecho de que la multiplicación `3 * 5` y `5 * 3` tengan el mismo resultado es solo una propiedad de la multiplicación, pero no tiene porque cumplirse para los monoides.

`Product` y `Sum`

Ya hemos visto una forma de que los números sean considerados monoides, con la función `*` y la identidad `1`. Resulta que no es la única forma de que los números formen un monoide. Otra forma sería utilizando la función binaria`+` y como identidad `0`:

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

La leyes de los monoides se cumplen. Si sumamos a un número 0, el resultado es ese número. Además la suma también es asociativa, así que no tenemos ningún problema. Entonces tenemos dos formas de que los números sean monoides, ¿cuál elegimos? Bueno, no tenemos porque elegir. Recuerda, cuando existen varias instancias de un tipo válidas para una misma clase de tipos, podemos utilizar `newtype` con ese tipo y crear una instancia para cada comportamiento.

En este caso el módulo `Data.Monoid` exporta dos tipos, llamados `Product`y `Sum`. `Product` se define así:

```
newtype Product a =  Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Simple, es solo un tipo `newtype` con un parámetro de tipo y algunas clases derivadas. Su instancia para la clase `Monoid` es esta:

```
instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

`mempty` es simplemente un `1` envuelto en el constructor `Product`. El patrón de `mappend` se ajusta al constructor `Product`, multiplica los dos números y devuelve el resultado como `Product`. Como puedes ver, existe una restricción de clase `Num a`. Esto quiere decir que `Product a` tendrá una instancia para `Monoid` siempre que `a` sea miembro de la clase `Num`. Para utilizar `Product a` como monoide, tenemos que introducir y extraer los valores:

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4 `mappend` Product 2
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
```

Es bonito como ejemplo de la clase de tipos `Monoid`, pero nadie en su sano juicio utilizaría esta forma para multiplicar números en lugar de escribir`3 * 9` y `3 * 1`. Aún así, dentro de poco veremos como estas instancias de `Monoid` que parece triviales ahora pueden ser muy útiles.

`Sum` se define como `Product` y su instancia es similar. Lo utilizamos del mismo modo:

```
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

Any y *All*

Otro tipo que puede comportarse como un monoide de dos formas diferentes y válidas es `Bool`. La primera forma es tener la función lógica *O* `||` como función binaria junto al valor `False` como identidad. La función lógica *O* devuelve `True` si alguno de sus dos parámetros es `True`, en caso contrario devuelve `False`. Así que si utilizamos `False` como valor identidad, la función binaria devolverá `False` si su otro parámetro es `False` y `True` si su otro parámetro es `True`. El constructor `newtype Any` tiene una instancia para `Monoid`. Se define así:

```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Y la instancia así:

```
instance Monoid Any where
        mempty = Any False
        Any x `mappend` Any y = Any (x || y)
```

La razón por la que se llama `Any` (*Algún*) es porque devuelve `True` si *alguno* de sus parámetros es `True`. Aunque tres o más valores `Bool` envueltos en `Any` sean reducidos con `mappend`, el resultado se mantendrá a `True` si alguno de ellos es `True`:

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny $ mempty `mappend` Any True
True
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True
ghci> getAny $ mempty `mappend` mempty
False
```

La otra forma de que `Bool` sea miembro de la clase `Monoid` es la contraría: tener `&&` como función binaría y `True` como valor identidad. La función lógica *Y* devuelve `True` solo si ambos parámetros son `True`. Aquí tienes la declaración de `newtype`:

```
newtype All = All { getAll :: Bool }
        deriving (Eq, Ord, Read, Show, Bounded)
```

Y la instancia es:

```
instance Monoid All where
        mempty = All True
        All x `mappend` All y = All (x && y)
```

Cuando utilizamos `mappend` con tipos `All`, el resultado será `True`solo si todos los valores son `True`:

```
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll $ mempty `mappend` All False
False
ghci> getAll . mconcat . map All $ [True, True, True]
True
ghci> getAll . mconcat . map All $ [True, True, False]
False
```

Al igual que la multiplicación y la suma, normalmente especificamos explícitamente la función binaria en lugar de introducir los datos en un tipo`newtype` para luego utilizar `mappend`. `mconcat` parece útil para`Any` y `All`, pero normalmente es más fácil usar las funciones `or` o`and`, las cuales toman una lista de `Bool` y devuelven `True` si hay algún `True` o `True` si todos los son, respectivamente.

El monoide `Ordering`

¿Recuerdas el tipo `Ordering`? Se utiliza como resultado al comparar cosas y tiene tres posibles valores: `LT`, `EQ` y `GT`, cuyo significado es*menor que*, *igual que* y *mayor que* respectivamente:

```
ghci> 1 `compare` 2
LT
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

Con las listas, los números, los valores booleanos simplemente era cuestión de buscar una función ya existente que mostrara un comportamiento de monoide. Con`Ordering` tenemos que buscar más detalladamente para encontrar el monoide, pero resulta que su instancia de `Monoid` es tan intuitiva como las otras que ya hemos visto:

```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```

La instancia funciona de este modo: cuando aplicamos `mappend` a dos valores`Ordering`, el valor de la izquierda se mantiene como resultado, a no ser que dicho valor sea `EQ`, en cuyo el resultado será el valor de la derecha. La identidad es `EQ`. A primera vista puede parecer un poco arbitrario, pero en realidad se asemeja a la forma en la que comparamos las palabras por orden alfabético. Comparamos las dos primeras letras y si son diferentes ya podemos decidir cual irá primero en el diccionario. Sin embargo, si las primeras dos letras son iguales, tenemos que comparar el siguiente par de letras y repetir el proceso.

Por ejemplo, si comparamos alfabéticamente las palabras `"le"` y `"la"`, primero comparamos las primeras letras de ambas palabras, al comprobar que son iguales continuamos con las segundas letras de cada palabra. Vemos que `'e'`es alfabéticamente mayor que `'a'` y de este modo obtenemos el resultado. Para aclarar porque `EQ` es la identidad, podemos ver que en caso de comparar la misma letra en la misma posición de ambas palabras, `EQ` no cambiaría el resultado de su orden alfabético. `"lxe"` sigue siendo alfabéticamente mayor que `"lxa"`.

Es importante tener en cuenta que la instancia de `Monoid` para`Ordering`, `x `mappend` y` no es igual `y `mappend` x`. Como el primer parámetro se mantiene como resultado a no ser que sea `Eq`, `LT `mappend`GT` devuelve `LT`, mientras que `GT `mappend` LT` devuelve `GT`:

```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

Vale, así que, ¿cuál es la utilidad de este monoide? Digamos que estamos escribiendo una función que toma dos cadenas, compara sus longitudes y devuelve un resultado del tipo `Ordering`. Pero si las cadenas son del mismo tamaño, en lugar de devolver `EQ`, las compara alfabéticamente. Una forma de escribir esto sería así:

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = let a = length x `compare` length y
                        b = x `compare` y
                    in  if a == EQ then b else a
```

Damos el nombre `a` al resultado de comparar las cadenas por sus longitudes y `b` al resultado de compararlas alfabéticamente. Si resulta que sus longitudes son idénticas, devolvemos `b`.

Pero ahora que sabemos que `Ordering` es un monoide, podemos reescribir esta función de una forma mucho más simple:

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (x `compare` y)
```

Vamos a probarla:

```
ghci> lengthCompare "zen" "peces"
LT
ghci> lengthCompare "zen" "pez"
GT
```

Recuerda, cuando utilizamos `mappend` el parámetro izquierdo será el resultado a no ser que sea igual a `Eq`, en cuyo caso el resultado será el parámetro derecho. Por esta razón ponemos la comparación que según nuestro criterio es más importante como primer parámetro. Si queremos expandir esta función para que también compare por el número de vocales a modo de segundo criterio más importante, simplemente tenemos que modificarla así:

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (vowels x `compare` vowels y) `mappend`
                    (x `compare` y)
    where vowels = length . filter (`elem` "aeiou")
```

Hemos creado una función auxiliar que toma una cadena y nos dice cuantas vocales tiene. Para ello filtra las letras que estén contenidas en la cadena`"aeiou"` y luego aplica `length`.

```
ghci> lengthCompare "zen" "anna"
LT
ghci> lengthCompare "zen" "ana"
LT
ghci> lengthCompare "zen" "ann"
GT
```

Genial. En el primer ejemplo vemos que si las longitudes de las cadenas son distintas devuelve `LT`, ya que la longitud de `"zen"` es menor que la de `"anna"`. En el segundo ejemplo, las longitudes son iguales, pero la segunda cadena tiene más vocales, así que devuelve `LT` de nuevo. En el tercer ejemplo, ambas cadenas tienen la misma longitud y el mismo número de vocales, así que se comparan alfabéticamente y `"zen"` gana.

El monoide `Ordering` es muy útil ya que nos facilita comparar cosas por varios criterios en un cierto orden, del más importante al menos importante.

El monoide `Maybe`

Vamos a ver varias formas en las que `Maybe a` puede actuar como un monoide y para que son útiles.

Una forma sería tratar `Maybe a` como monoide solo si su parámetro de tipo `a` es también un monoide, de forma que `mappend` se implemente utilizando `mappend` con los valores contenidos en `Just`. Utilizamos `Nothing` como identidad, de modo que si uno de los dos valores que pasamos a `mappend` es `Nothing`, el resultado será el otro valor. Así sería la declaración de la instancia:

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Fíjate en la restricción de clase. Dice que `Maybe a` tendrá una instancia de `Monoid` solo si `a` posee una instancia de `Monoid`. Si aplicamos `mappend` a algo y a `Nothing`, el resultado será ese algo. Si aplicamos `mappend` a dos valores `Just`, se aplicará `mappend` al contenido de ambos valores y el resultado de dicha operación se introducirá en un valor `Just`. Somos capaces de realizar esta operación porque la restricción de clase nos asegura que el tipo que contiene los valores `Just` posee una instancia de `Monoid`.

```
ghci> Nothing `mappend` Just "andy"
Just "andy"
ghci> Just LT `mappend` Nothing
Just LT
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

Este monoide es útil cuando estamos trabajando con resultados de cómputos que pueden fallar. Gracias a esta instancia, no nos tenemos que preocupar por comprobar si los cómputos han fallado y por lo tanto son `Nothing` o bien son `Just`. Simplemente debemos tratarlos como valores normales.

Pero, ¿qué sucede si el contenido de `Maybe` no forma parte de la clase `Monoid`? Si te fijas, en la declaración de la instancia anterior, el único lugar donde necesitábamos que los contenidos fueran monoides era cuando los dos parámetros de `mappend` eran `Just`. Pero si desconocemos si los contenidos son monoides o no, no podemos utilizar `mappend` con ellos, así que, ¿qué podemos hacer? Bueno, una de las cosas que podemos hacer es descartar el segundo valor y quedarnos con el primero. Por este motivo existe el tipo `First a` y esta es su definición:

```
newtype First a = First { getFirst :: Maybe a }
    deriving (Eq, Ord, Read, Show)
```

Tomamos un tipo `Maybe a` y lo envolvemos en un `newtype`. La instancia de `Monoid` sería así:

```
instance Monoid (First a) where
    mempty = First Nothing
    First (Just x) `mappend` _ = First (Just x)
    First Nothing `mappend` x = x
```

Tal y como hemos dicho. `mempty` es simplemente `Nothing` dentro del constructor `newtype First`. Si el primer parámetro de `mappend` es un valor `Just` ignoramos el segundo parámetro. Si el primer parámetro es `Nothing`, entonces damos el segundo parámetro como resultado, independientemente de que sea `Nothing` o `Just`:

```
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getFirst $ First (Just 'a') `mappend` First Nothing
Just 'a'
```

`First` es útil cuando tenemos un montón de valores `Maybe` y solamente queremos saber si alguno de ellos es `Just`. La función `mconcat` será útil en esos momentos:

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

Si queremos que un monoide `Maybe a` conserve el segundo parámetro cuando se aplica `mappend` sobre dos valores `Just` en lugar del primero,`Data.Monoid` proporciona el tipo `Last a` que funciona igual que`First a`, solo que el último valor no `Nothing` se mantiene como resultado cuando se utiliza `mappend` o `mconcat`:

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

Utilizando monoides para plegar estructuras de datos

Una de las formas más interesante de utilizar los monoides es para que nos ayuden a la hora de definir pliegues sobre estructuras de datos. Hasta ahora solo hemos utilizados pliegues sobre listas, pero las listas no son el único tipo de dato que se puede plegar. Podemos definir pliegues sobre casi cualquier tipo de estructura. Lo árboles, por ejemplo, encajan perfectamente con este concepto.

Debido a que existen tantos tipos de datos que funcionan perfectamente con los pliegues, utilizamos la clase de tipos `Foldable`. De la misma forma `Functor` es para cosas que pueden ser mapeadas, `Foldable` es para cosas que puede ser plegadas. Se encuentra en `Data.Foldable` y como exporta funciones con nombres iguales a funciones residentes en `Prelude`, se mejor importarlo cualificado:

```
import qualified Foldable as F
```

Para evitar sobre-esforzar nuestros queridos dedos, hemos elegido importarlo con el nombre `F`. ¿Y bien? ¿cuáles son las funciones que define esta clase de tipos? Bueno, básicamente son `foldr`, `foldl`, `foldr1` y`foldl1`. ¿Cómo? Ya conocemos estas funciones así que, ¿qué tienen diferente? Vamos a comparar los tipos de `foldr` de `Prelude` y de`foldr` de `Foldable` para ver las diferencias:

```
ghci> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
ghci> :t F.foldr
F.foldr :: (F.Foldable t) => (a -> b -> b) -> b -> t a -> b
```

Mientras que `foldr` toma una lista y la pliega, el `foldr` de`Data.Foldable` toma cualquier tipo que pueda ser plegado , ¡no solo listas! Como era de esperar, ambas funciones se comportan igual con las listas.

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

Bien, ¿qué otras estructuras soportan pliegues? Pues nuestro querido `Maybe` por ejemplo.

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

Pero plegar valores `Maybe` no es que sea especialmente interesante, ya que cuando realiza el pliegue actúa como si se tratara de una lista con un solo elemento en caso de que valor sea `Just` o como una lista vacía si el valor es `Nothing`. Vamos a examinar una estructura de datos que sea más compleja.

¿Recuerdas la estructura de datos de árbol que utilizamos en el capítulo de"*Creando nuestros propios tipos de datos*"? Lo definimos así:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

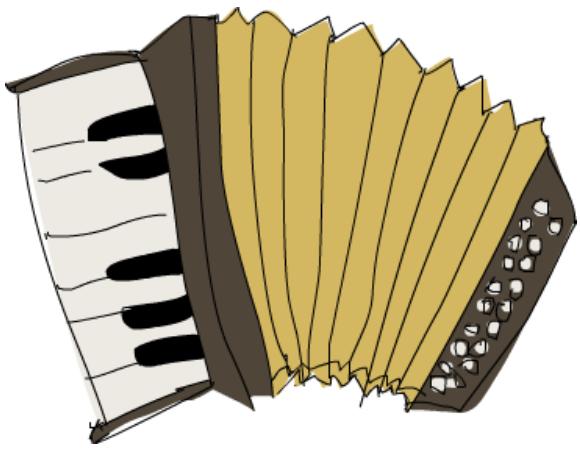
Dijimos que un árbol es o bien en un árbol vacío que no contiene ningún valor o bien un nodo que contiene un valor y también otros dos árboles. Después de definirlo, creamos su instancia para la clase `Functor` y con ella ganamos la habilidad de usar `fmap` con estos árboles. Ahora vamos a crear la instancia de `Foldable` de forma que podamos plegar estos árboles. Una forma de crear la instancia de `Foldable` para un determinado constructor de tipos es simplemente implementar la función `foldr` para él. Pero existe otra forma, normalmente mucho más sencilla, y es implementar la función `foldMap`, que también forma parte de la clase de tipos `Foldable`. La función `foldMap` tiene la siguiente declaración de tipo:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

El primer parámetro es una función que toma un valor del tipo que la estructura de datos plegable contiene (denominada aquí `a`) y devuelve un monoide. El segundo parámetro es una estructura plegable que contiene valores del tipo `a`. Mapea esa función sobre la estructura de datos plegable, produciendo de esta forma una estructura plegable que contiene monoides. Luego, aplicando `mappend` entre estos monoides, los reduce todos en un solo valor. Esta función puede parece algo rara ahora mismo, pero veremos que es realmente fácil de implementar. Otra cosa que también es interesante es que cuando implementemos estas función no necesitaremos hacer nada más para implementar el resto de la instancia `Foldable`. Así que simplemente tenemos que implementar la función `foldMap` para un tipo, y obtenemos automáticamente `foldr` y `foldl`.

Aquí tienes como definimos la instancia de `Foldable` para `Tree`:

```
instance F.Foldable Tree where
    foldMap f Empty = mempty
    foldMap f (Node x l r) = F.foldMap f l `mappend`
                             f x           `mappend`
                             F.foldMap f r
```



Si tenemos una función que toma un elemento del árbol y devuelve un monoide, ¿cómo reducimos el árbol entero a un único monoide? Cuando aplicamos `fmap`a un árbol, aplicamos la función que estamos mapeando a un nodo y luego mapeamos recursivamente esa misma función sobre el sub-árbol izquierdo y el sub-árbol derecho. En este caso nuestra tarea no consiste únicamente en mapear una función, sino que también tenemos que reducir todos los resultados en un único monoide utilizando `mappend`. Primero consideramos el caso de un árbol vacío, un triste y solitario árbol que no contienen ningún valor ni ningún sub-árbol. Como no contiene ningún valor no podemos aplicar la

función binaria del monoide, así que nos limitamos a decir que el árbol está vacío devolviendo el valor `mempty`.

El caso de un nodo no vacío es algo más interesante. Contiene dos sub-árboles y también un valor. En este caso, aplicamos recursivamente la función`foldMap` junto a `f` al sub-árbol izquierdo y derecho. Recuerda, la función `foldMap` devuelve un único valor, un monoide. También aplicamos la función `f` al valor contenido en el nodo. De este modo, ahora tenemos tres monoides (dos de los sub-árboles y otro de aplicar `f` al nodo actual) y solo tenemos que reducirlos a un único valor. Para lograrlo utilizamos la función `mappend`, primero con el valor del sub-árbol izquierdo, luego con el valor del nodo y para terminar con el valor del sub-árbol derecho.

Ten en cuenta que no somos los encargados de crear la función que toma un valor y devuelve un monoide. Recibimos esa función como parámetro de`foldMap` y todo lo que debemos hacer es decidir donde vamos a aplicarla y como reducir los monoides a un único valor.

Ahora que tenemos una instancia de `Foldable` para el tipo árbol, obtenemos`foldr` y `foldl` automáticamente. Considera este árbol:

```
testTree = Node 5
            (Node 3
                (Node 1 Empty Empty)
                (Node 6 Empty Empty)
            )
            (Node 9
                (Node 8 Empty Empty)
                (Node 10 Empty Empty)
            )
```

Tiene un `5` como raíz y luego su nodo izquierdo tiene un `3` con un `1` a su izquierda y un `6` a su derecha. El nodo raíz de la derecha contiene un`9` junto con un `8` a su izquierda y un `10` a la derecha. Gracias a la instancia de `Foldable` ahora podemos utilizar todos los pliegues que usábamos con las listas:

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

`foldMap` no solo es útil para crear nuevas instancias de `Foldable`. También es útil para reducir una estructura a un único valor monoidal. Por ejemplo, si queremos saber si algún número de un árbol es igual a `3`podemos hacer lo siguiente:

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

Aquí, `\x -> Any $ x == 3` es una función que toma un número y devuelve un valor monoidal, en concreto un `Bool` dentro de un `Any`. `foldMap`aplica esta función a todos los elementos del árbol y luego reduce todos los resultados monoidales a un único valor monoidal. Si hacemos esto:

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

Todos los nodos del árbol contendrían el valor `Any False` después de que la función lambda se aplicara sobre ellos. Para que `mappend` sobre valores del tipo `Any` devuelva `True` al menos uno de sus parámetros debe ser`True`. Por este motivo el resultado que obtenemos al final es `False`, y tiene sentido ya que no hay ningún valor mayor que `15` en el árbol.

También podemos convertir el árbol en una lista fácilmente. Para ello utilizamos `foldMap` con la función `\x -> [x]`. Al proyectar la función sobre el árbol obtenemos un árbol que contendrá listas unitarias. Luego se aplicará `mappend` sobre todas esas listas de forma que obtendremos una única lista que contendrá todos los valores del árbol original.

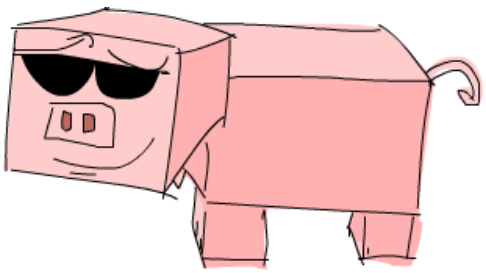
```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

Lo mejor de todo es que este truco no se limita únicamente a los árboles, funciona para cualquier tipo miembro de la clase `Foldable`.

Un puñado de mónadas

La primera vez que hablamos de los funtores, vimos que son un concepto útil para los valores que se pueden mapear. Luego, llevamos ese concepto un poco más lejos y vimos los funtores aplicativos, que nos permite ver los valores de ciertos tipos de datos como una especia de valores con un contexto de forma que podemos aplicar funciones normales sobre ellos mantenido dicho contexto.

En este capítulos hablaremos de las mónadas, que simplemente son una versión ampliada de los funtores aplicativos, de la misma forma que los funtores aplicativos son una versión ampliada de los funtores.



Cuando hablamos de los funtores vimos que es era posible mapear funciones sobre varios tipos de datos. Para lograrlo utilizábamos la clase de tipos`Functor`. Dada una función del tipo `a -> b` y un dato del tipo `f a` nos preguntábamos cómo mapeamos esta función sobre el dato de forma que obtuviésemos un resulto con el tipo `f b`. Vimos como mapear funciones sobre datos del tipo `Maybe a`, del tipo `[a]`, `IO a`, etc. Incluso vimos como mapear funciones `a -> b` sobre funciones `r -> a` de forma que el resultado daba funciones del tipo `r -> b`. Para contestar a la pregunta que nos hacíamos de como mapear una función sobre un dato, lo único que tenemos que hacer es mirar el tipo de `fmap`:

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

Y luego hacer que funcione con el tipo de datos para el que estamos creando la instancia de `Functor`.

Luego vimos que era posible mejorar los funtores. Decíamos: ¡Hey! ¿qué pasa si tenemos una función `a -> b` dentro del valor de un funtor? Como por ejemplo, `Just (*3)`, y queremos aplicarla a `Just 5`. ¿Qué pasaría si en vez de aplicarla a `Just 5` la aplicamos a `Nothing`? ¿O si tenemos`[(*2),(+4)]` y queremos aplicarla a `[1,2,3]`? ¿Cómo hacemos para que funcione de forma general? Para ello utilizamos la clase de tipos`Applicative`.

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

También vimos que podíamos tomar un valor normal e introducirlo dentro de un tipo de datos. Por ejemplo, podemos tomar un `1` en introducirlo en un tipo `Maybe` de forma que resulte en `Just 1`. O incluso podríamos crear un `[1]`. O una acción de E/S que no realizara nada y devolviera un `1`. La función que realiza esta acción es `pure`.

Como ya dijimos, un valor aplicativo puede verse como un valor dentro de un contexto. Un valor *adornado* en términos técnicos. Por ejemplo, el carácter`'a'` es un simple carácter normal, mientras que `Just 'a'` tiene añadido un cierto contexto. En lugar de un `Char` tenemos un `Maybe Char`, que nos dice que su valor puede ser un carácter o bien la ausencia de un carácter.

También es genial ver como la clase de tipos `Applicative` nos permite utilizar funciones normales con esos contextos de forma que los contextos se mantengan. Observa:

```
ghci> (*) <$> Just 2 <*> Just 8
Just 16
ghci> (++) <$> Just "klingon" <*> Nothing
Nothing
ghci> (-) <$> [3,4] <*> [1,2,3]
[2,1,0,3,2,1]
```

Estupendo, ahora que los tratamos con valores aplicativos, los valores`Maybe a` representan cómputos que pueden fallar, `[a]` representan cómputos que tienen varios resultados (cómputos no deterministas), `IO a`representan valores que tienen efectos secundarios, etc.

Las mónadas son una extensión natural de los funtores aplicativos y tratan de resolver lo siguiente: si tenemos un valor en un cierto contexto, `m a`, ¿cómo podemos aplicarle una función que toma un valor normal `a` y devuelve un valor en un contexto? Es decir, ¿cómo podemos aplicarle una función del tipo `a -> m b`? Básicamente lo que queremos es esta función:

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

Si tenemos un valor adornado y una función que toma un valor y devuelve un valor adornado, ¿cómo pasamos el primer valor adornado a la función?. Esta será la pregunta principal que nos haremos cuando trabajemos con las mónadas. Escribimos `m a` en lugar de `f a` ya que `m` representa mónadas, aunque las mónadas no son más que funtores aplicativos que soportan la función`>>=`. Llamamos *lazo* a la función `>>=`.

Cuando tenemos un valor normal `a` y una función normal `a -> b` es muy fácil pasar ese valor a la función. Simplemente hay que aplicar la función a ese valor de forma normal. Pero cuando estamos trabajando con valores que vienen dentro de un cierto contexto, tenemos que tomarnos un tiempo para ver como estos valores adornados se pasan a las funciones y para ver como se comportan. No te preocupes, verás que es muy fácil.

Manos a la obra con Maybe

Ahora que ya tenemos una pequeña idea del cometido de la mónadas, vamos a expandirla en detalle.

Para sorpresa de nadie, `Maybe` es una mónada, así que vamos a explorarlo un poco más a ver si podemos combinar lo que ya sabemos con las mónadas.

Nota
Llegados a este punto, asegúrate de que entiendes los *funtores aplicativos*. Será más fácil si sabes como funcionan varias instancias de`Applicative` y que tipo de cómputos representan, ya que las mónadas no son más que una expansión de los funtores aplicativos.

Un valor de tipo `Maybe a` representa un valor del tipo `a` dentro del contexto de que ocurra un posible fallo. Un valor `Just "dharma"` representa que la cadena `"dharma"` está presente mientras que `Nothing` representa su ausencia, o si vemos la cadena como el resultado de un cómputo, significará que dicho cómputo ha fallado.

Cuando hablamos de `Maybe` como funtor vimos que cuando mapeamos una función sobre él con `fmap`, se mapea solo cuando es un valor `Just`, de otro modo `Nothing` se mantiene como resultado ya que no hay nada sobre lo que mapear.

Como esto:

```
ghci> fmap (++"!") (Just "wisdom")
Just "wisdom!"
ghci> fmap (++"!") Nothing
Nothing
```

Como funtor aplicativo funciona de forma similar. Sin embargo, los funtores aplicativos también poseen funciones dentro de los funtores. `Maybe` es un funtor aplicativo de forma que cuando aplicamos `<*>` con una función contenida en un `Maybe` a un valor contenido en un `Maybe`, ambos deben ser valores `Just` para que el resultado sea también un valor `Just`, en caso contrario el resultado será `Nothing`. Tiene sentido ya que si no tenemos o bien la función o bien el valor, no podemos crear un resultado a partir de la nada, así que hay que propagar el fallo:

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

Cuando utilizamos el estilo aplicativo con funciones normales para que actúen con valores del tipo `Maybe` es similar. Todos los valores deben ser`Just` si queremos que el resultado también lo sea.

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

Y ahora vamos a ver como podríamos implementar `>>=` para `Maybe`. Como ya hemos dicho, `>>=` toma un valor monádico y una función que toma un valor normal y devuelve otro valor monádico, de forma que aplica esta función al valor monádico. ¿Cómo consigue hacerlo si la función solo toma valores normales? Bueno, para lograrlo hay que tomar en cuenta el contexto de ese valor monádico.

En este caso, `>>=` tomará un valor con el tipo `Maybe a` y una función de tipo `a -> ``Maybe b` y de alguna forma aplicará esta función para dar como resultado `Maybe b`. Para imaginarnos como se hace, podemos apoyarnos en lo que ya sabemos de los funtores aplicativos. Digamos que tenemos una función del tipo `\x -> Just (x+1)`. Toma un número, le añade `1` y lo introduce en un `Just`:

```
ghci> (\x -> Just (x+1)) 1
Just 2
ghci> (\x -> Just (x+1)) 100
Just 101
```

Si le pasaramos como parámetro `1` devolvería `Just 2`. Si le pasaramos`100` devolvería `Just 101`. Simple. Ahora viene lo bueno: ¿cómo pasamos un dato del tipo `Maybe` a esta función? Si pensamos en `Maybe` como un funtor aplicativo contestar a esta pregunta es bastante fácil. Si le pasamos un valor `Just`, toma el valor que contiene y le aplica la función. Si le pasamos `Nothing`, mmm, bueno, tenemos la función pero no tenemos nada que pasarle. En este caso vamos a hacer lo mismo que hicimos anteriormente y diremos que el resultado será `Nothing`.

En lugar de llamar a esta función `>>=`, vamos a llamarla `applyMaybe` por ahora. Toma un `Maybe a` y una función que devuelve un `Maybe b` y se las ingenia para aplicar esa función a `Maybe a`. Aquí está la función:

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

Vale, ahora vamos a jugar un poco con ella. La utilizamos de forma infija de forma que el valor `Maybe` estará en la parte izquierda y la función a aplicar en la parte derecha:

```
ghci> Just 3 `applyMaybe` \x -> Just (x+1)
Just 4
ghci> Just "smile" `applyMaybe` \x -> Just (x ++ " :)")
Just "smile :)"
ghci> Nothing `applyMaybe` \x -> Just (x+1)
Nothing
ghci> Nothing `applyMaybe` \x -> Just (x ++ " :)")
Nothing
```

En este ejemplo vemos que cuando utilizamos `applyMaybe` con un valor`Just` y una función, la función simplemente se aplica al valor contenido en`Just`. Cuando la utilizamos con un valor `Nothing`, el resultado final es`Nothing`. ¿Qué pasa si la función devuelve un `Nothing`? Vamos ver:

```
ghci> Just 3 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Just 3
ghci> Just 1 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Nothing
```

Justo como imaginábamos. Si el valor monádico de la izquierda es `Nothing`, el resultado final es `Nothing`. Y si la función de la derecha devuelve`Nothing`, el resultado será de nuevo `Nothing`. Es muy parecido a cuando utilizabamos `Maybe` como funtor aplicativo y obteniamos como resultado`Nothing` si en algún lugar había un `Nothing`.

Parace que para `Maybe`, hemos averiguado como tomar un valor decorado y pasarlo a una función que toma un parámetro normal y devuelve otro valor decorado. Lo hemos conseguido teniendo en cuenta que los valores del tipo`Maybe` representan cómputo que pueden fallar.

Seguramente te este preguntado: ¿y esto es útil? Puede parecer que los funtores aplicativos son más potentes que las mónadas, ya que los funtores aplicativos permiten tomar una función normal y hacer que opere sobre valores con un cierto contexto. Veremos que las mónadas pueden hacer exactamente lo mismo ya que son una versión mejorada de los funtores aplicativos, pero también veremos que pueden hacer más cosas que los funtores aplicativos no pueden hacer.

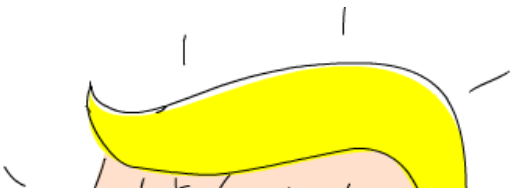
Volvermos con `Maybe` en un momento, pero primero, vamos a ver la clase de tipos que define las mónadas.

La clase de tipos de las mónadas

De la misma forma que los funtores tienen una clase `Functor` y que los funtores aplicativos tienen una clase `Applicative`, las mónadas vienen con su propia clase de tipos: `Monad` ¡Wau! ¿Quíen lo hubiera imaginado? Así es como luce su definición:

```
class Monad m where
    return :: a -> m a

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

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

Empecemos por la primera línea. Dice `class Monad m where`. Pero espera, ¿no hemos dicho que las mónadas no son más que funtores aplicativos ampliados? ¿No debería haber una resitricción de clase como`class (Applicative m) = > Monad m where` de

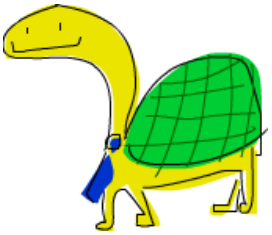


forma que el tipo tenga que ser un funtor aplicativo primero antes de ser una mónada? Bueno, debería, pero cuando se creo Haskell, la gente que lo creo no pensó que los funtores aplicativos encajarían bien en Haskell así que no aparece. Pero ten seguro que cada mónada es un funtor aplicativo, incluso aunque la declaración de la clase`Monad` diga lo contrario.

La primera función que define la clase de tipos `Monad` es `return`. Es lo mismo que `pure` pero con un nombre diferente. Su tipo es`(Monad m) => a -> m a`. Toma un valor y lo introduce en el contexto por defecto que pueda albergar dicho valor. En otras palabras, toma un valor y lo introduce en una mónada. Siempre hace lo mismo que la función `pueda` de la clase de tipos `Applicative`, por lo que ya estmos familiarizados al uso de `return`. Ya hemos utilizado `return` cuando trabajamos con la E/S. La utilizabamos para crear una acción de E/S que no hiciera nada salvo contener un valor. Con `Maybe` toma un valor y lo introduce en un valor `Just`.

Nota

Recordatorio: `return` no se parece en nada al `return` de la mayoría de los otro lenguajes de programación. No termina la ejecución ni nada por el estilo, simplemente toma un valor normal y lo introduce en un contexto.



La siguiente función es `>>=` o lazo. Es como la aplicación de funciones, solo que en lugar de tomar un valor y pasarlo a una función normal, toma un valor monádico (es decir, un valor en un cierto contexto) y lo pasa a una función que toma un valor normal pero devuelve otro valor monádico.

A continuación tenemos `>>`. No le vamos a prestar mucha ateción ahora mismo ya que viene con una implementación por defecto y prácticamente nunca tendremos que implementarla cuando creemos instancias de `Monad`.

La función final de la clase de tipos `Monad` es `fail`. Nunca la utilizaremos explícitamente en nuestro código. En cambio, Haskell la utilizará para permitir fallos en una construción sintáctica para las mónadas que veremos más adelante. No tenemos que preocuparnos demasiado con `fail` ahora mismo.

Ahora que ya sabemos como luce la clase de tipos `Monad`, vamos a ver como es la instancia de `Maybe` para la clase `Monad`:

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

`return` es lo mismo que `pure`, no hay que pensar mucho. Hacemos exactamente lo mismo que hacíamos con `Applicative`, introducimos un valor en `Just`.

La función `>>=` es exactamente igual `applyMaybe`. Cuando le pasamos un valor del tipo `Maybe a` a esta función, tenemos en cuenta el contexto y devolvemos `Nothing` si el valor a la izquierda es `Nothing` ya que no existe forma posible de aplicar la función con este valor. Si es un valor`Just` tomamos lo que hay dentro de él y aplicamos la función.

Podemos probar un poco `Maybe` como mónada:

```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

Nada nuevo o emocionante en la primera línea ya que ya hemos usado `pure`con `Maybe` y sabemos que `return` es igual que `pure` solo que con otro nombre. La siguientes dos líneas muestran como funciona `>>=` un poco más.

Fíjate en como hemos pasado `Just 9` a la función `\x -> return (x*10)`,`x` toma el valor `9` dentro de la función. Parece como si fueramos capaces de extraer el valor de un `Maybe` sin utilizar un ajuste de patrones. Y aún así no perdemos el contexto de los tipo `Maybe`, porque cuando es `Nothing`, el resultado de `>>=` será `Nothing` también.

En la cuerda floja

Ahora que ya sabemos como parar un valor del tipo `Maybe a` a una función del tipo `a -> Maybe b` teniendo en cuenta el contexto de un posible fallo, vamos a ver como podemos usar `>>=` repetidamente para manejar varios valores `Maybe a`.

Pierre ha decidido tomar un descanso en su trabajo en la piscifactoria e intentar caminar por la cuerda floja. No lo hace nada mal, pero tiene un problema: ¡los pájaros se posan sobre su barra de equilibrio! Aterrizan y se toman un pequeño respiro, hablan con sus respectivos amigos ovíparos y luego se marchan en busca de algo de comida. Ha Pierre no le importaría demasiado si el número de pájaros que se posan en cada lado de la barra fuera el mismo. Sin embargo, a menudo, todos los pájaros se posan en el mismo lado y desequilibran a Pierre tirándolo de la cuerda de forma embarazosa (utiliza un red de seguridad obviamente).

Digamos que matiene el equilibrio si el número de pájaros posados a la izquierda y a la derecha de la barra no difere en más de tres. Así que si hay un pájaro en la parte derecha y otros cuatro pájaros en la parte izquierda no pasa nada. Pero si un quinto pájaro aterriza en la parte derecha pierde el quilibrio y cae.

Vamos a simular un grupo de pájaros que aterrizan o inician el vuelo desde la barra y ver si Pierre sigue sobre la barra tras un número de eventos relacionados con estas aves. Por ejemplo, queremos saber que le pasará a Pierre si primero llega un pájaro al lado izquierdo de la barra, luego cuatro pájaros más se posan sobre la parte derecha y luego el pájaro de la izquierda decide volar de nuevo.

Podemos representar la barra con un par de enteros. El primer componente indicará el número de pájaros a la izquierda mientras que el segundo indicará el número de pájaros de la derecha:

```
type Birds = Int
type Pole = (Birds,Birds)
```

Primero creamos un sinónimo para `Int`, llamado *pájaros* (`Birds`), ya que estamos utilizando enteros para representar el número de pájaros. Luego creamos otro sinónimo de tipos `(Birds, Birds)` y lo llamamos *barra*(`Pole`).

A continuación creamos una función que toma un número de pájaros y los posa sobre un determinado lado de la barra. Aquí están las funciones:

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)

landRight :: Birds -> Pole -> Pole
landRight n (left,right) = (left,right + n)
```

Bastante simple. Vamos a probarlas:

```
ghci> landLeft 2 (0,0)
(2,0)
ghci> landRight 1 (1,2)
(1,3)
ghci> landRight (-1) (1,2)
(1,1)
```

Para hacer que los pájaros vuelen simplemente tenmos que pasarles a estas funciones un número negativo. Como estas funciones devuelven un valor del tipo `Pole`, podemos encadenarlas:

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0,0)))
(3,1)
```

Cuando aplicamos la función `landLeft 1` a `(0, 0)` obtenemos `(1, 0)`. Luego aterrizamos un pájaro sobre el lado derecho, por lo que obtenemos`(1, 1)`. Para terminar aterrizamos dos pájaros más sobre el lado izquierdo, lo cual resulta en `(3, 1)`. Aplicamos una función a algo escribirendo primero el nombre de la función y luego sus parámetros, pero en este caso sería mejor si la barra fuera primero y luego las funciones de aterrizar. Si creamos una función como:

```
x -: f = f x
```

Podríamos aplicar funciones escribiendo primero el parámetro y luego el nombre de la función:

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0, 0) -: landLeft 2
(2,0)
```

Utilizando esto podemos aterrrizar varios pájaros de un forma mucho más legible:

```
ghci> (0, 0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

¡Genial! Es ejemplo es equivalente al ejemplo anterior en el que aterrizamos varias aves en la barra, solo que se ve más limpio. Así es más obvio que empezamos con `(0, 0)` y luego aterrizamos un pájaro sobre la izquierda, otro sobre la derecha y finalmente dos más sobre la izquierda.

Hasta aquí bien, pero, ¿qué sucede si aterrizan diez pájaros sobre un lado?

```
ghci> landLeft 10 (0,3)
(10,3)
```

¿Diez pájaros en la parte izquierda y solo tres en la derecha? Seguro que Pierre ya debe estar volando por los aires en esos momentos. En este ejemplo es bastante obvio pero, ¿y si tenemos una secuencia como esta?:

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

A primera vista puede parecer que todo esta bien pero si seguimos los pasos, veremos que en un determinado momento hay cuatro pájaros a la derecha y ninguno a la izquierda. Para arreglar esto debemos darle una vuelta de tuerca a las funciones `landLeft` y `landRight`. A partir de lo que hemos aprendido queremos que estas funciones sean capaces de fallar. Es decir, queremos que devuelvan una barra si Pierre consigue mantener el equilibrio pero que fallen en caso de que Pierre lo pierda. ¡Y qué mejor manera de añadir el contexto de un posible fallo a un valor que utilizar `Maybe`! Vamos a reescribir estas funciones:

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)
    | abs ((left + n) - right) < 4 = Just (left + n, right)
    | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right)
    | abs (left - (right + n)) < 4 = Just (left, right + n)
    | otherwise                    = Nothing
```

En lugar de devolver un `Pole` estas funciones devuelven un `Maybe Pole`. Siguen tomando el número de pájaros y el estado de la barra anterior, pero ahora comprueban si el número de pájaros y la posición de estos es suficiente como para desquilibrar a Pierre. Utilizamos guardas para comprabar si diferencia entre el número de pájaros en cada lado es menor que cuatro. Si lo es devuelve una nueva barra dentro de un `Just`. Si no lo es, devuelven`Nothing`.

Vamos a jugar con estas pequeñas:

```
ghci> landLeft 2 (0,0)
Just (2,0)
ghci> landLeft 10 (0,3)
Nothing
```

¡Bien! Cuando aterrizamos pájaros sin que Pierre pierda el equilibrio obtenemos una nueva barra dentro de un `Just`. Pero cuando unos cunatos pájaros de más acaban en un lado de la barra obtenemos `Nothing`. Esto esta muy bien pero ahora hemos perido la posibilidad de aterrizar pájaros de forma repetiva sobre la barra. Ya no podemos usar `landLeft 1 (landRight 1 (0,0))`ya que cuando aplicamos `landRight 1` a `(0, 0)` no obtenemos un `Pole`, sino un `Maybe Pole`. `landLeft 1` toma un `Pole` y no un`Maybe Pole`.

Necesitamos una forma de tomar un `Maybe Pole` y pasarlo a una función que toma un `Pole` y devuelve un `Maybe Pole`. Por suerte tenemos `>>=`, que hace exáctamen lo que buscamos para `Maybe`. Vamos a probarlo:

```
ghci> landRight 1 (0,0) >>= landLeft 2
Just (2,1)
```

Recuerda, `landLeft 2` tiene un tipo `Pole -> Maybe Pole`. No podemos pasarle directamente un valor del tipo `Maybe Pole` que es el resultado de`landRight 1 (0, 0)`, así que utilizamos `>>=` que toma un valor con un determinado contexto y se lo pasa a `landLeft 2`. De hecho `>>=` nos permite tratar valores `Maybe` como valores en un contexto si pasamos`Nothing` a `landLeft 2`, de forma que el resultado será `Nothing` y el fallo ser propagará:

```
ghci> Nothing >>= landLeft 2
Nothing
```

Gracias a esto ahora podemos encadenar varios aterrizajes que pueden consguir tirar a Pierre ya que `>>=` nos permite pasar valores monádicos a funciones que toman valores normales.

Aquí tienes una secuencia de aterrizajes:

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

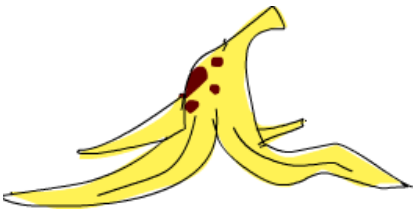
Al principio hemos utilizado `return` para insertar una barra dentro de un`Just`. Podríamos haber aplicado `landRight 2` directamente a `(0, 0),hubiéramos llegado al mismo resultado, pero de esta forma podemos utilizar``>>=` para cada función de forma más consistente. Se pasa `Just (0, 0)` a`landRight 2`, lo que devuelve `Just (0, 2)`. Luego se le pasa este valor a `landLeft 2` obteniendo `Just (2, 2)` y así sucesivamente.

Recuerda el ejemplo que dijimos que tiraría a Pierre:

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

Como vemos no simula la interacción con las aves correctamente ya que en medio la barra ya estaría volando por los aires pero el resultado no lo refleja. Pero ahora vamos a probar a utilizar la aplicación monádica (`>>=`) en lugar de la aplicación normal:

```
ghci> return (0,0) >>= landLeft 1 >>= landRight 4 >>= landLeft (-1) >>= landRight (-2)
Nothing
```



Perfecto. El resultado final representa un fallo, que es justo lo que esperamos. Vamos a ver como se consigue este resultado. Primero, `return`introduce `(0, 0)` en el contexto por defecto, convirtiéndolo en `Just (0, 0)`. Luego sucede `Just (0,0) >>= landLeft 1`. Como`Just (0,0)` es un valor `Just`, `landLeft 1` es aplicado a `(0, 0)`, obteniendo así `Just (1, 0)` ya que Pierre sigue manteniendo el equilibrio. Luego nos encontramos con `Just (1,0) >>= landRight 4` lo cual resulta en`Just (1, 4)` ya que Pierre sigue manteniendo el equilibrio, aunque malamente. Se aplica `landLeft (-1)` a `Just (1, 4)`, o dicho de otra forma, se computa `landLeft (-1) (1,4)`. Ahora, debido a como funciona`landLeft`, esto devuelve `Nothing` porque nuestro esta volando por los aires en este mismo momento. Ahora que tenemos `Nothing` como resultado, éste se pasado a `landRight (-2), pero como es un valor ``Nothing`, el resultado es automáticamente `Nothing` ya que no existe ningún valor que se puede aplicar a `landRight (-2)`.

No podíamos haber conseguido esto utilizando solo `Maybe` como funtor aplicativo. Si lo intentas te quedarás atascado, porque los funtores aplicativos no permiten que los valores aplicativos interactuen con los otros lo sufiente. Pueden, como mucho, ser utilizados como parámetros de una función utilizando el estilo aplicativo. Los operadores aplicativos tomarán los resultados y se los pasarán a la función de forma apropiada para cada funto aplicativo y luego obtendrán un valor aplicativo, pero no existe ninguna interacción entre ellos. Aquí, sin embargo, cada paso depende del resultado anterior. Por cada aterrizaje se examina el resultado anterior y se comprueba que la barra está balanceada. Esto determina si el aterrizaje se completará o fallará.

Podemos divisar una función que ignora el número de pájaros en la barra de equilibrio y simplemente haga que Pierre caiga. La llamaremos `banana`:

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

Ahora podemos encadenar esta función con los aterrizajes de las aves. Siempre hara que Pierre se caiga ya que ignora cualquier cosa que se le pasa y devuelve un fallo. Compruebalo:

```
ghci> return (0,0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

El valor `Just (1, 0)` se le pasa a `banana`, pero este produce `Nothing`, lo cual hace que el resultado final sea `Nothing`. Menuda suerte.

En lugar de crear funciones que ignoren el resultado y simplemente devuelvan un valor monádico, podemos utilizar la función `>>` cuya implementación por defecto es esta:

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

Normalmente, si pasamos un valor a una función que toma un parámetro y siempre devuelve un mismo valor por defecto el resultado será este valor por defecto. En cambio con la mónadas también debemos conseiderar el contexto y el siguinificado de éstas. Aquí tienes un ejemplo de como funciona `>>` con`Maybe`:

```
ghci> Nothing >> Just 3
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

Si reemplazamos `>>` por `>>= \_ ->` es fácil de ver lo que realmente sucede.

Podemos cambiar la función `banana` por `>>` y luego un `Nothing`:

```
ghci> return (0,0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

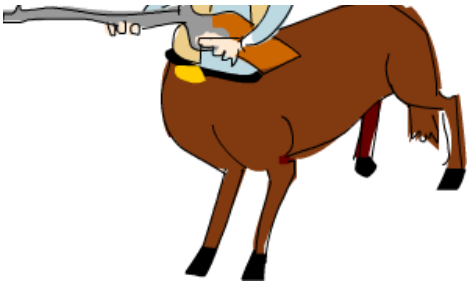
Ahí lo tienes, ¡garantizamos que Pierre se va ir al suelo!

También vale la pena echar un vistazo a como se veria esto si no hubiesemos tratado los valores `Maybe` como valores en un contexto y no hubiersemos pasado las parámetros a las funciones como hemos hecho. Así es como se vería una serie de aterrizajes:

```
routine :: Maybe Pole
routine = case landLeft 1 (0,0) of
    Nothing -> Nothing
    Just pole1 -> case landRight 4 pole1 of
        Nothing -> Nothing
        Just pole2 -> case landLeft 2 pole2 of
            Nothing -> Nothing
            Just pole3 -> landLeft 1 pole3
```

Aterrizamos un pájaro y comprobamos la posibiliadad de que que ocurra un fallo o no. En caso de fallo devolvemos `Nothing`. En caso contrario aterrizamos unos cuantos pájaros más a la derecha y volemos a comprobar lo mismo una y otra vez. Convertir esto es un limpia concatenación de aplicaciones monádicas con `>>=` es un ejemplo clásico de porque la mónada

`Maybe` nos ahorra mucho tiempo cuando tenemos una secuecnia de cómputos que dependen del resultado de otros cómputos que pueden fallar.



Fíjate en como la implementación de `>>=` para `Maybe` interpreta exactamente la lógica de que en caso encontrarnos con un `Nothing`, lo devolvemos como resultado y en caso contrario continuamos con lo que hay dentro de `Just`.

En esta sección hemos tomado varias funciones que ya teniamos y hemos visto que funcionan mejor si el valor que devuelven soporta fallos. Conviertiendo estos valores en valores del tipo `Maybe` y cambiando la aplicación de funciones normal por `>>=` obtenemos un mecanismo para manejar fallos casi de forma automática, ya que se supone `>>=` preserva el contexto del valor que se aplica a una función. En este caso el contexto que tenían estos valores era la posibiliadad de fallo de forma que cuando aplicábamos funciones sobre estos valores, la posibilidad de fallo siempre era tomada en cuenta.

La notación Do

Las mónadas son tan útiles en Haskell que tienen su propia sintaxis especial llamada notación `do`. Ya nos hemos topado con la notación `do` cuando reliazabamos acciones de E/S y dijimos que servia para unir varias de estas acciones en una sola. Bueno, pues resulta que la notación `do` no solo funciona con `IO` sino que puede ser utilizada para cualquier mónada. El principio sigue siendo el mismo: unir varios valores monádicos en secuencia. Vamos a ver como funiona la notación `do` y porque es útil.

Considera el siguiente ejemplo familiar de una aplicación monádica:

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

Pasamos un valor monádico a una función que devuelve otro valor monádico. Nada nuevo. Fíjate que en el ejemplo anterior, `x` se convierte en `3`, es decir, una vez dentro de la función lambda, `Just 3` pasa a ser un valor normal en vez de un valor monádico. Ahora, ¿qué pasaría si tuviésemos otro`>>=` dentro de la función?

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

¡Wau, un `>>=` anidado! En la función lambda interior, simplemente pasamos`Just !` a `\y -> Just (show x ++ y)`. Dentro de esta lambda, `y` se convierte en `"!"`. `x` sigue siendo el `3` que obtuvimos de la lambda exterior. Esto se parece a la siguiente expresión:

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

La diferencia principal entre ambas es que los valores de la primera son valores monádicos. Son valores con el contexto de un posible fallo. Podemos remplazar cualquier valor por un fallo:

```
ghci> Nothing >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Nothing >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Nothing))
Nothing
```

En la primera línea, pasamos `Nothing` a una función y naturalmente resulta en `Nothing`. En la segunda línea pasamos `Just 3` a la función de forma que `x` se convierte en `3`, pero luego pasamos `Nothing` a la función lambda interior así que el resultado

es también `Nothing`. Todo esto es parecido a ligar nombres con ciertos valores utilizando las expresiones`let`, solo que en lugar de valores normales son valores monádicos.

El siguiente ejemplo ilustra esta idea. Vamos a escribir lo mismo solo que cada valor `Maybe` esté en una sola línea:

```
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))
```

En lugar de escribir todos estas funciones lambdas, Haskell nos proporciona la sintaxis `do` que nos permite escribir el anterior trozo de código como:

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

Puede parecer que hemos ganado la habilidad de cosas de valores `Maybe` sin tener que preocuparnos por comprobar en cada paso si dichos valores son valores `Just` o valores `Nothing` ¡Genial! Si alguno de los valores que intentamos extraer es `Nothing`, la expresión `do` entera se reducirá a `Nothing`. Estamos extrayendo sus (probablemente existentes) valores y dejamos a `>>=` que se preocupe por el contexto de dichos valores. Es importante recordar que la notación `do` es solo una sintaxis diferente para encanedar valores monádicos.

En una expresión `do` cada línea es un valor monádico. Para inspecionar el resultado de una línea utilizamos `<-`. Si tenemos un `Maybe String` y le damos una variable utilizando `<-`, esa variable será del tipo `String`, del mismo modo que cuando utilizábamos `>>=` para pasar valores monádicos a las funciones lambda. El último valor monádico de una expresión, en este caso`Just (show x ++ y)`, no se puede utilizar junto a `<-` porque no tendría mucho sentido traducimos de nuevo la expresión `do` a una ecandención de aplicaciones `>>=`. Esta última línea será el resultado de unir toda la expresión `do` en un único valor monádico, teniendo en cuenta el hecho de que puede ocurrir un fallo en cualquiera de los pasos anteriores.

Por ejemplo:

```
ghci> Just 9 >>= (\x -> Just (x > 8))
Just True
```

Como el parámetro a la izquierda de `>>=` es un valor `Just`, la función lambda es aplicada a `9` y el resultado es `Just True`. Si reescribimos esto en notación `do` obtenemos:

```
marySue :: Maybe Bool
marySue = do
    x <- Just 9
    Just (x > 8)
```

Si comparamos ambas es fácil deducir porque el resultado de toda la expresión`do` es el último valor monádico.

La función `routine` que escribimos anteriormente también puede ser escrita con una expresión `do`. `landLeft` y `landRight` toman el número de pájaros y la barra para producir una nueva barra dentro de un valor `Just`, a no ser que nuestro funambulista se caiga y produzca `Nothing`. Utilizamos`>>=` porque cada uno de los pasos depende del anterior y cada uno de ellos tiene el contexto de un posible fallo. Aquí tienes dos pájaros posandose en lado izquierdo, luego otros dos pájaros posandose en lado derecho y luego otro más aterrizando en la izquierda:

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    second <- landRight 2 first
    landLeft 1 second
```

Vamos a ver si funciona:

```
ghci> routine
Just (3,2)
```

¡Lo hace! ¡Genial! Cuando creamos esta función utilizando `>>=`, utilizábamos cosas como `return (0,0) >>= landLeft 2`, porque `landLeft 2`es una función que devuelve un valor del tipo `Maybe`. Sin embargo con las expresiones `do`, cada línea debe representar un valor monádico. Así que tenemos que pasar explícitamente cada `Pole` anterior a las funciones`landLeft` y `landRight`. Si examinamos las variables a las que ligamos los valores `Maybe`, `start` sería `(0,0)`, `first` sería `(2,0)` y así sucesivamente.

Debido a que las expresiones `do` se escriben línea a línea, a mucha gente le puede parecer código imperativo. Pero lo cierto es que son solo secuenciales, de forma que cada línea depende del resultado de las líneas anteriores, junto con sus contextos (en este caso, dependen de si las anterioeres fallan o no).

De nuevo, vamos a volver a ver como sería este código si no tuvieramos en cuenta los aspectos monádicos de `Maybe`:

```
routine :: Maybe Pole
routine =
    case Just (0,0) of
        Nothing -> Nothing
        Just start -> case landLeft 2 start of
            Nothing -> Nothing
            Just first -> case landRight 2 first of
                Nothing -> Nothing
                Just second -> landLeft 1 second
```

Fíjate como en caso de no fallar, la tupla dentro de `Just (0,0)` se convierte en `start`, el resultado de `landLeft 2 start` se convierte en `first`, etc.

Si queremos lanzar a Pierre una piel de plátano en notación `do` solo tenemos que hacer lo siguiente:

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    Nothing
    second <- landRight 2 first
    landLeft 1 second
```

Cuando escribirmos una línea en la notación `do` sin ligar el valor monádico con `<-`, es como poner `>>` después de ese valor monádico cuyo reulstado queremos que ignore. Secuenciamos el valor monádico pero ignoramos su resultado ya que no nos importa y es más cómodo que escribir `_ <- Nothing`, que por cierto, es lo mismo.

Cuando utilizar la notación `do` y cuando utilizar `>>=` depende de ti. Creo que este ejemplo se expresa mejor escribiendo explícitamente los `>>=`ya que cada paso depende específicamente del anterior. Con la notación `do` tenemos que especificar en que barra van a aterrizar los pájaros incluso aunque siempre aterrizen en la barra anterior.

En la notación `do`, cuando ligamos valore monádicos a variables, podemos utilizar ajustes de patrones de la misma forma que los usábamos con las expresiones `let` o con los parámetros de las funciones. Aquí tienes un ejemplo de uso de ajuste de patrones dentro de una expresión `do`:

```
justH :: Maybe Char
justH = do
    (x:xs) <- Just "hello"
    return x
```

Hemos ajustado un patrón para obtener el primer carácter de la cadena`"hello"` y luego lo devolvemos como resultado. Así que `JustH` se evalua a `Just 'h'`.

¿Qué pasaria si este ajuste fallara? Cuando un ajuste de patrones falla en una función se utiliza el siguiente ajuste. Si el ajuste falla en todos los patrones de una función, se lanza un error y el programa podría terminar. Por otra parte si el ajuste falla en una expresión `let`, se lanza un error directamente ya que no existe ningún mecanismo que no lleve a otro patrón que ajustar. Cuando un ajuste falla dentro de una expresión `do` se llama a la función `fail`. Ésta es parte de la clase de tipos `Monad` y nos permite ver este fallo como un fallo en el contexto del valor monádico en lugar de hacer que el programa termine. Su implementación por defecto es:

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

Así que por defecto hace que el programa termine, pero las mónadas que incorporan un contexto para un posible fallo (como `Maybe`) normalmente implementan el suyo propio. En `Maybe` se implementa así:

```
fail _ = Nothing
```

Ignora el mensaje de error y devuelve `Nothing`. Así que cuando un ajuste falla dentro de un valor `Maybe` que utiliza una expresión `do`, el valor entero se reduce a `Nothing`. Suele ser preferiable a que el programa termine. Aquí tienes una expresión `do` con un patrón que no se ajustará y por tanto fallará:

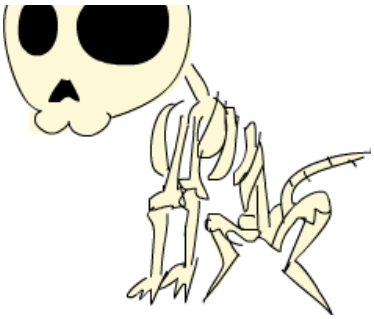
```
wopwop :: Maybe Char
wopwop = do
    (x:xs) <- Just ""
    return x
```

El ajuste falla, así que sería igual a remplazar toda la línea por`Nothing`. Vamos a probarlo:

```
ghci> wopwop
Nothing
```

Este fallo en el ajuste de un patrón genera un fallo en el contexto de nuestra mónada en lugar de generar un fallo en el programa, lo cual es muy elegante.

La mónada lista



Hasta ahora hemos visto como los valores del tipo `Maybe` pueden verse como valores en un contexto de un posible fallo y que podemos incorportar el tratamiento de estos posibles fallos utilizando `>>=` para pasar los parámetros a las funciones. En esta sección vamos a echar un vistazo a como podemos utilizar los aspectos monádicos de las listas llevanso así el no determinsmo a nuestro código de forma legible.

Ya hemos hablado de como las listas representan valores no deterministas cuando se utilizan como funtores aplicativos. Un valor como `5` es determinista. Tiene un único valor y sabemos exactamente cual es. Por otra parte, un valor como `[3,8,9]` consiste en varios resultados, así que lo podemos ver como un valor que en realidad es varios valores al mismo tiempo.

Al utilizar las listas como funtores aplicativos vemos fácilmente este no determinismo:

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

Todas la posibles soluciones de multiplicar los elementos de la izquierda por los elementos de la derecha aparecen en la lista resultado. Cuando trabajamos con el no determinismo, exsiten varias opciones que podemos tomar, así que básicamente probamos todas ellas y por lo tanto el resultado también otro valor no determinista, solo que con unos cuantos valores más.

Este contexto de no determinismo se translada a las mónadas fácilmente. Vamos a ver como luce la instancia de `Monad` para las listas:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

`return` es lo mismo que `pure`, así que ya estamos familiarizados con ella. Toma un valor y lo introducie en el mínimo contexto por defecto que es capaz de albergar ese valor. En otras palabras, crea una lista que contiene como único elemento dicho valor. Resulta útil cuando necesitmos que un valor determinista interactue con otros valores no deterministas.

Para entender mejor como funciona `>>=` con las listas veremos un ejemplo de su uso. `>>=` toma un valor con un contexto (un valor monádico) y se lo pasa a una función que toma valores normales y devuelve otro valor en el mismo contexto. Si esta función devolviera un valor normal en luegar de un valor monádico, `>>=` no sería muy útil ya que depués de usarlo perderíamos el contexto. De cualquier modo, vamos vamos a intentar pasar un valor no determinista a una función:

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Cuando utilizamos `>>=` con `Maybe`, el valor monádico se pasaba a la función teniendo en cuenta la existencia de un posible fallo. Aquí `>>=`se preocupa del no determinismo por nosotros. `[3,4,5]` es un valor no determinista y se lo hemos pasado a otra función que devuelve valores no deterministas también. El resultado final también es no determinista y contiene los posibles resultados de aplicar la función `\x -> [x,-x]` a todos los elementos de `[3,4,5]`. Esta función toma un número y produce dos resultados: uno negado y otro igual que el original. De esta forma cuando utilizamos `>>=` para pasar la lista a esta función todos los números son negados pero también se mantienen los originales. La `x` de la función lambda toma todos los posibles valores de la lista que pasamos como parámetro.

Para ver como se consigue este resultado solo tenemos que ver la implementación. Primero, empezamos con la list `[3,4,5]`. Luego mapeamos la función lambda sobre ella y obtenemos el siguiente resultado:

```
[[3,-3],[4,-4],[5,-5]]
```

La función lambda se aplica a cada elemento por lo que obtenemos una lista de listas. Para terminar simplemente concatenamos las listas y punto final ¡Acabamos de aplicar un función no determinista a una valor no determinista!

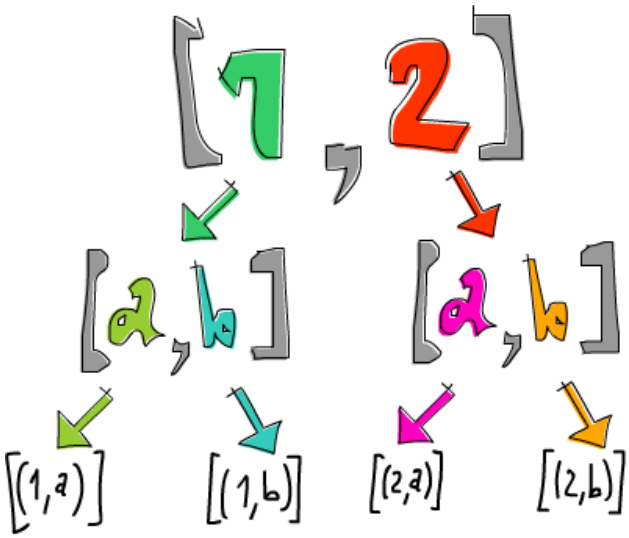
El no determinismo también soporta la existencia de fallos. La lista vacía`[]` es muy parecido a `Nothing` ya que ambos representan la ausencia de un resultado. Por este motivo la función `fail` se define simplemente con la lista vacía. El mensaje de error se ignora.

```
ghci> [] >>= \x -> ["bad","mad","rad"]
[]
ghci> [1,2,3] >>= \_ -> []
[]
```

En la primera línea se pasa una lista vacía a la función lambda. Como la lista no tienen ningún elemento, no podemos pasar nada a la función así que el resultado final es también la lista vacía. Es similiar a pasar `Nothing` a una función. En la segunda línea, cada elemento de la lista se pasa a la función, pero estos elementos son ignorados y la función simplemente devuelve una lista vacía. Como la función falla para todos los elementos de la lista, el resultado final es la lista vacía.

Del mismo modo que pasaba con los valores del tipo `Maybe`, podemos concatenar varios `>>=` propagando así el no determinismo:

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```



Los elemenots de lista `[1,2]` se ligan a `n` y los elementos de`['a','b']` se ligan a `ch`. Luego, hacemos `return (n,ch)` (o`[(n,ch)]`), lo que significa que tomamos una dupla `(n,ch)` y la introducimos en el contexto mínimo por defecto. En este caso, se crea la lista más pequeña posible que pueda albergar `(n,ch)` como resultado de forma que posea tan poco no determinismo como sea posible. Dicho de otro modo, el efecto del contexto es mínimo. Lo que estamos implementando es: para cada elemento en `[1,2]` y para cada elemento de `['a','b']` producimos una dupla para combinación posible.

En términos generales, como `return` lo único que hace es introducir un valor en el contexto mínimo, no posee ningún efecto extra (como devolver un fallo en `Maybe` o devolver en un valor aún menos determinista en caso de las listas) sino que sólamete toma un valor como resultado.

Nota
Cuando tenemos varios valores no deterministas interactuando, podemos ver su cómputo como un árbol donde cada posible resultado representa una rama del árbol.

Aquí tienes la expresión anterior escrita con notación `do`:

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
```

```
    ch <- ['a','b']
    return (n,ch)
```

Así parece más obvio que `n` toma cada posible valor de `[1,2]` y que`ch` toma cada posible valor de `['a','b']`. Del mismo modo que con`Maybe`, estamos extrayendo valores normales de un valor monádico y dejamos que `>>=` se preocupe por el contexto. El contexto en este caso es el no determinismo.

Cuando vemos las listas utilizando la notación `do` puede que nos recuerde a algo que ya hemos visto. Mira esto:

```
ghci> [ (n,ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

¡Sí! ¡Listas por comprensión! Cuando utilizábamos la notación `do`, `n`tomaba cada posible elemento de `[1,2]` y `ch` tomaba cada posible elemento de `['a','b']` y luego introducíamos `(n,ch)` en el contexto por defecto (una lista unitaria) para devolverlo como resultado final sin tener que introducir ningún tipo de no determinismo adicional. En esta lista por comprensión hacemos exactamente lo mismo, solo que no tenemos que escribir`return` al final para dar como resultado `(n,ch)` ya que la lista por comprensión se encarga de hacerlo.

De hecho, las listas por comprensión no son más que una alternativa sintáctica al uso de listas como mónadas. Al final, tanto las listas por comprensión como la notación `do` se traduce a una concatenación de `>>=` que representan el no determinismo.

Las listas por comprensión nos perminten filtrar la lista. Por ejemplo, podemos filtrar una lista de número para quedarnos únicamente con los números que contengan el dígito `7`:

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

Aplicamos `show` a `x` para convertir el número en una cadena y luego comprobamos si el carácter `'7'` froma parte de en esa cadena. Muy ingenioso. Para comprender como se traduce estos filtros de las listas por comprensión a la mónada lista tenemos que ver la función `guard` y la clase de tipos `MonadPlus`. La clase de tipos `MonadPlus` representa mónadas que son también monoides. Aquí tienes la definición:

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

`mzero` es un sinónimo del `mempty` que nos encontramos en la clase`Monoid` y `mplus` correponde con `mappend`. Como las listas también son monoides a la vez que mónadas podemos crear una isntancia para esta clase de tipos:

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

Para las listas `mzero` representa un cómputo no determinista que no devuelve ningún resultado, es decir un cómputo que falla. `mplus` une dos valores no deterministas en uno. La función `guard` se define así:

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

Toma un valor booleano y si es `True`, introduce `()` en el mínimo contexto por defecto. En caso contrario devuleve un valor monádico que representa un fallo. Aquí la tienes en acción:

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

Parece interesante pero, ¿es útil? En la mónada lista utilizamos esta función para filtrar una series de cómputos no deterministas. Observa:

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

El resultado es el mismo que la lista por comprensión anterior. ¿Cómo consigue `guard` este resultado? Primero vamos a ver se utiliza `guard`junto a `>>`:

```
ghci> guard (5 > 2) >> return "cool" :: [String]
["cool"]
ghci> guard (1 > 2) >> return "cool" :: [String]
[]
```

Si el predicado de `guard` se satisface, el resultado es una lista con una tupla vacía. Luego utilizamos `>>` para ignorar esta tupla vacía y devolver otra cosa como resultado. Sin embargo, si `guard` falla, no alcanzaremos el `return` ya que si pasamos una lista vacía a una funcón con `>>=` el resultado siempre será una lista vacía. `guard` simplemente dice: si el predicado es `False` entonces devolvemos un fallo, en caso contrario devolvemos un valor que contiene un resultado ficticio `()`. Esto permite que el encadenamiento continue.

Así sería el ejemplo anterior utilizando la notación `do`:

```
sevensOnly :: [Int]
sevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    return x
```

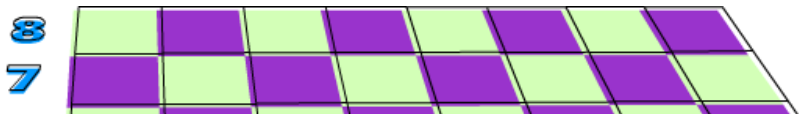
Si hubiéramos olvidado devolver `x` como resultado final con `return`, la lista resultante sería una lista de tuplas vacías en lugar de una lista de enteros. Aquí tienes de nuevo la lista por comprensión para que compares:

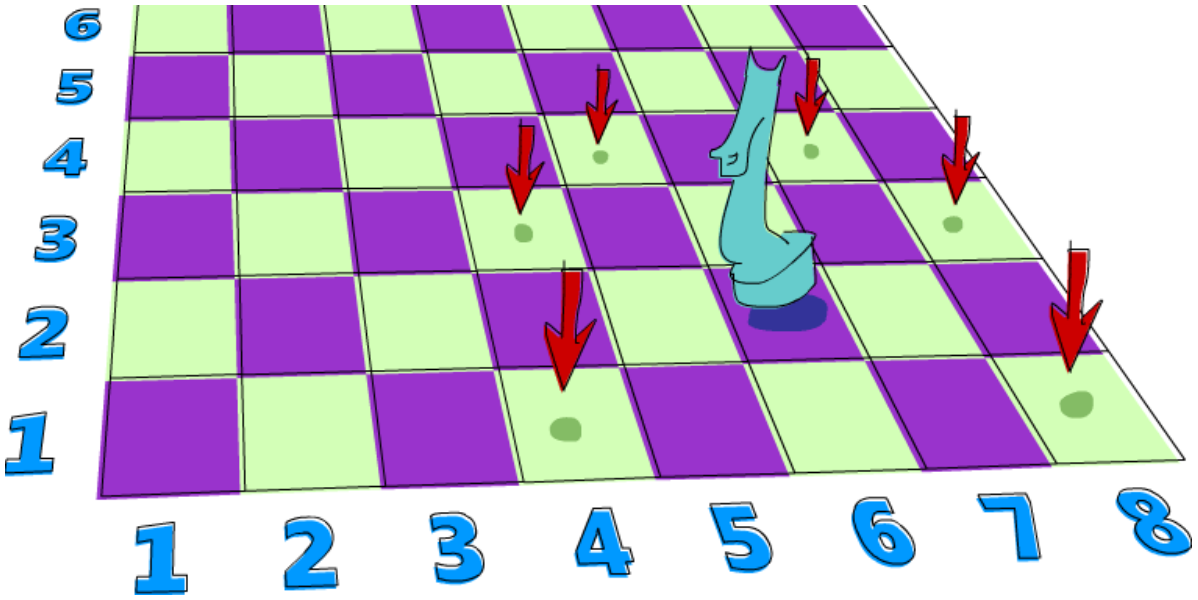
```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

Filtrar una lista por comprensión es igual que usar `guard`.

El salto del caballo

Vamos a ver un problema que tiende a resolverse utilizando no determinismo. Digamos que tenemos un tablero de ajedrez y como única pieza un caballo. Queremos saber si el caballo peude alcanzar una determinada posición en tres movimientos. Utilizaremos una dupla de números para representar la posición del caballo en el tablero. El primer número representará la columna en la que está el caballo y el segundo representará la fila.





Vamos a crear un sinónimo de tipo para representar la posición actual del caballo:

```
type KnightPos = (Int,Int)
```

Digamos que el caballo empieza en `(6,2)` ¿Puede alcanzar `(6,1)` en solo tres movimientos? Vamos a ver. Si empezamos en `(6,2)`, ¿cuál sería el mejor movimiento a realizar? Ya se, ¡Todos! Tenemos el no determinismo a nuestra disposición, así que en lugar de decidirnos por un movimiento, hagámoslos todos. Aquí tienes una función que toma la posición del caballo y devuelve todos las posibles posiciones en las que se encontrará depués de moverse.

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
               ]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    return (c',r')
```

El caballo puede tomar un paso en horizontal o vertical y otros dos pasos en horizontal o vertical pero siempre haciendo un movimiento horizontal y otro vertical. `(c',r')` toma todos los valores de los elementos de la lista y luego `guard` se encarga de comprobar que la nueva posicion permanece dentro del tablero. Si no lo está, produce una lista vacía y por lo tanto no se alcanza `return (c',r')` para esa posición.

También se puede escribir esta función sin hacer uso de la mónada lista, aunque lo acabamos de hacer solo por diversión. Aquí tienes la misma función utilizando `filter`:

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = filter onBoard
    [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
    ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
    ]
    where onBoard (c,r) = c `elem` [1..8] && r `elem` [1..8]
```

Ambas son iguales, así que elige la que creas mejor. Vamos a probarla:

```
ghci> moveKnight (6,2)
[(8,1),(8,3),(4,1),(4,3),(7,4),(5,4)]
ghci> moveKnight (8,1)
[(6,2),(7,3)]
```

¡Funciona perfectamente! Toma una posición y devuelve todas las siguientes posiciones de golpe. Así que ahora que tenemos la siguiente posición de forma no determinista, solo tenemos que aplicar `>>=` para pasársela a `moveKnight`. Aquí tienes una función que toma una posición y devuelve todas las posiciones que se pueden alcanzar en tres movimientos:

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
    first <- moveKnight start
    second <- moveKnight first
    moveKnight second
```

Si le pasamos `(6,2)`, el resultado será un poco grande porque si existe varias formas de llegar a la misma posición en tres movimientos, tendremos varios elementos repetidos. A continuación sin usar la notación `do`:

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

Al utiliza `>>=` obtenemos todos los posibles movimientos desde el inicio y luego cuando utilizamos `>>=` por segunda vez, para cada posible primer movimiento calculamos cada posible siguiente movimiento. Lo mismo sucede para el tercer movimiento.

Introducir un valor en el contexto por defecto utilizando `return` para luego pasarlo como parámetro utilizando `>>=` es lo mismo que aplicar normalemente la función a dicho valor, aunque aquí lo hemos hecho de todas formas.

Ahora vamos a crear una función que tome dos posiciones y nos diga si la última posición puede ser alcanzada con exáctamente tres pasos:

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

Generamos todas las posibles soluciones que se pueden generar con tres pasos y luego comprobamos si la posición destino se encuentra dentro de estas posibles soluciones. Vamos a ver si podemos alcanzar `(6,1)` desde `(6,2)`en tres movimientos:

```
ghci> (6,2) `canReachIn3` (6,1)
True
```

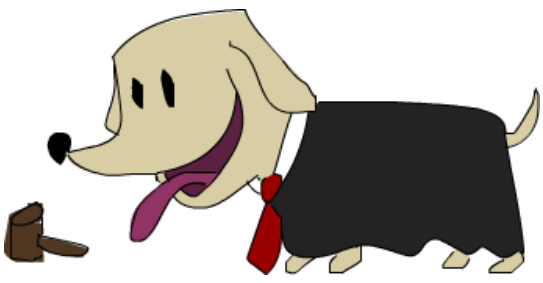
¡Sí! ¿Y de `(6,2)` a `(7,3)`?

```
ghci> (6,2) `canReachIn3` (7,3)
False
```

¡No! Como ejercicio, puedes intentar modificar esta función para que cuando se pueda alcanzar esta posición te diga que pasos debes seguir. Luego, veremos como modificar esta función de forma que también pasemos como parámetro el número de pasos.

Las leyes de las mónadas

De la misma forma que lo funtores aplicativos, a la vez que lo funtores normales, las mónadas vienen con una serie de leyes que todas las mónadas que se precien deben cumplir. Solo porque algo tenga una instancia de la clase`Monad` no significa que sea una mónada, solo significa que ese algo tiene una instancia para la clase `Monad`. Para que un tipo sea realmente una mónada debe satisfacer las leyes. Estas leyes nos permiten asumir muchas cosas acerca del comportamiento del tipo.



Haskell permite que cualquier tipo tenga una instancia de cualquier clase de tipos siempre que los tipos concuerden. No puede comprobar si las leyes de las mónadas se cumplen o no, así que si estamos creando una instancia para la clase `Monad`, tenemos que estar lo suficientemente seguros de que la mónada satisface las leyes para ese tipo. Los estar seguros de que los tipos que vienen en la biblioteca estándar cumplen estas leyes, pero luego, cuando creemos nuestras própias mónadas, tendremos que comprobar manualmente si se cumplen las leyes o no. No te asuste, no son complicadas.

Identidad por la izquierda

La primera ley establece que tomamos un valor, lo introducimos en el contexto por defecto utilizando `return` y luego pasamos el resultado a una función utilizando `>>=`, el resultado debe ser igual que aplicar la función directamente a ese valor. Informalmente:

- `return x >>= f` es exactamente lo mismo que `f x`.

Si vemos los valores monádicos como valores con un cierto contexto y`return` toma un valor y lo introduce en el contexto mínimo por defecto que puede albergar ese valor, tiene sentido que, como ese contexto en realidad es mínimo, al pasar el valor monádico a una función no debe haber mucha diferencia con aplicar la función a un valor normal, y de hecho, es exactamente lo mismo.

Para la mónada `Maybe`, `return` se define como `Just`. La mónada`Maybe` trata acerca de posibles fallos, así que si tenemos un valor y lo introducimos en dicho contexto, tiene sentido tratar este valor como cómputo correcto, ya que, bueno, sabemos cual es ese valor. Aquí tienes un par de usos de `return`:

```
ghci> return 3 >>= (\x -> Just (x+100000))
Just 100003
ghci> (\x -> Just (x+100000)) 3
Just 100003
```

En cambio para la mónada lista, `return` intruce un valor en una lista unitaria. La implementación de `>>=` para las listas recorre todos los elementos de la lista y les aplica una función, pero como solo hay un elemento en la lista, es lo mismo que aplicar la función a ese valor:

```
ghci> return "WoM" >>= (\x -> [x,x,x])
["WoM","WoM","WoM"]
ghci> (\x -> [x,x,x]) "WoM"
["WoM","WoM","WoM"]
```

Dijimos que para la mónada `IO`, `return` simplemente creaba una acción que no tenia ningún efecto secundario y solo albergaba el valor que pasábamos como parámetro. Así que también cumple esta ley.

Identidad por la derecha

La segunda ley establece que si tenemos un valor monádico y utilizamos `>>=`para pasarselo a `return`, el resultado debe ser el valor monádico original. Formalemente:

- `m >>= return` es igual que `m`.

Esta ley puede parecer un poco menos obvia que la primera, pero vamos a echar un vistazo para ver porque se debe cumplir. Pasamos valores monádicos a las funciones utilizando `>>=`. Estas funciones toman valores normales y devuelven valores monádicos. `return` es una también es una de estas funciones. Como ya sabemos, `return` introduce un valor en el contexto mínimo que pueda albergar dicho valor. Esto quiere decir que, por ejemplo para `Maybe`, no introduce ningún fallo; para las listas, no introduce ningún no determinismo adicional. Aqui tienes una prueba con algunas mónadas:

```
ghci> Just "move on up" >>= (\x -> return x)
Just "move on up"
ghci> [1,2,3,4] >>= (\x -> return x)
[1,2,3,4]
ghci> putStrLn "Wah!" >>= (\x -> return x)
Wah!
```

Si echamos un vistazo más de cerca al ejemplo de las listas, la implementación de `>>=` para las listas es:

```
xs >>= f = concat (map f xs)
```

Así que cuando pasamos `[1,2,3,4]` a `return`, primero `return` se mapea sobre `[1,2,3,4]`, devolviendo `[[1],[2],[3],[4]]` y luego se concatena esta lista obteniendo así la original.

La identida por la izquierda y la identadad por la derecha son leyes que establecen el comportamiento de `return`. Es una función importante para convertir valores normales en valores monádicos y no sería tan útil si el valor monádico que produciera hicera mucha más cosas.

Asociatividad

La última ley de las mónadas dice que cuando tenemos una cadena de aplicaciones funciones monádicas con `>>=`, no importa el orden en el que estén anidadas. Escrito formalmente:

- `(m >>= f) >>= g` es igual a `>>= (x -> f x >>= g)`.

Mmm... ¿Qué esta pasando aquí? Tenemos un valor monádico, `m` y dos funciones monádica `f` y `g`. Hacemos `(m >>= f) >>= g`, es decir, pasamos `m` a `f`, lo cual devuelve un valor monádico. Luego pasamos ese valor monádico a `g`. En la expresión `m >>= (\x -> f x >>= g)` tomamos un valor monádico y se lo pasamos a una función que pasa el resultado de`f x` a `g`. Quizá no es fácil ver como ambas expresiones son iguales, así que vamos a ver un ejemplo para aclarár las dudas.

¿Recuerdas cuando el funambulista Pierra caminaba sobre una cuerda con ayuda de una barra de equilibrio? Para simular el aterrizaje de los pájaros sobre esta barra de equilibrio utilizábamos una cadena de funciones que podían fallar:

```
ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

Empezábamos con `Just (0,0)` y luego pasábamos este valor a la siguiente función monádica, `landRight 2`. El resultado de ésta era otro valor monádico que pasábamos a la siguiente función de la cadena y así sucesivamente. Si mostramos la asociatividad de forma explícita, la expresión quedaría así:

```
ghci> ((return (0,0) >>= landRight 2) >>= landLeft 2) >>= landRight 2
Just (2,4)
```

Pero también podemos esxpresarlo así:

```
return (0,0) >>= (\x ->
landRight 2 x >>= (\y ->
landLeft 2 y >>= (\z ->
landRight 2 z)))
```

`return (0,0)` es lo mismo que `Just (0,0)` y cuando se lo pasamos a la función lambda, `x` se convierte en `(0,0)`. `landRight` toma un número de pájaros y una barra (una dupla de números) y eso es lo que le pasamos. Devuelve `Just (0,2)`

y cuando se lo pasamos a la siguiente función lambda,`y` es `(0,2)`. Continua hasta el último aterrizaje de pájaros que produce`Just (2,4)`, que de hecho es el resultado final de la expresión.

Resumiendo, no importa como anides el paso de valores monádicos, lo que importa es su significado. Otra forma de ver esta ley sería: consideremos la composición de dos funciones, `f` y `g`. La composición de funciones se implementa como:

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

El tipo de `g` es `a -> b` y el de `f` es `b -> c`, y las unimos en una nueva función con tipo `a -> c` cuyo parámetro será pasado entre las funciones anteriores. Y ahora, ¿qué pasaria si estas dos funciones fueran monádicas? es decir ¿qué pasaria si estas funciones devolvieran valores monádicos? Si tuvieramos una función del tipo `a -> m b`, no podríamos pasar su resultado directamente a una función del tipo `b -> m c`, ya que esta función solo acepta valores normales y no monádicos. Sin embargo podemos utilizar `>>=` para poder permitirlo. Así que si utilizamos `>>=`, podemos definir la composición de dos funciones monádicas como:

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

Ahora podemos componer nuevas funciones monádicas a partir de otras:

```
ghci> let f x = [x,-x]
ghci> let g x = [x*3,x*2]
ghci> let h = f <=< g
ghci> h 3
[9,-9,6,-6]
```

Genial ¿Y qué tiene que ver esto con la ley de asociatividad? Bueno, cuando vemos la ley como una ley de composiciones, ésta dice que`f <=< (g <=< h)` debe ser igual a `(f <=< g) <=< h`. Es otra forma de decir que para las mónadas, no importa el orden del anidamiento.

Si traducimos las dos primeras leyes para que utilicen `<=<`, entonces, la primera ley dice que para cada función monádica `f`, `f <=< return`es lo mismo que `f` y la ley de identidad por la derecha dice que`return <=< f` es también igual a `f`.

Es parecido a lo que ocurre con las funciones normales, `(f . g) . h` es lo mismo que `f . (g . h)`, `f . id` es igual a `f` y `id . f` es también igual a `f`.

En este capítulo hemos visto las bases de la mónadas y hemos aprendido a utilizar las mónadas `Maybe` y las listas. En el siguiente capítulo, echaremos un vistazo a un puñado más de mónadas y también aprenderemos como crear nuestras propias mónadas.

Unas cuantas mónadas más

Hemos visto como podemos utilizar las mónadas para tomar valores con un cierto contexto y aplicarles funciones utilizando `>>=` o la notación `do`, lo cual nos permite centrarnos en los valores en si mientras que el contexto se trata automáticamente.

Ya hemos conocemos la mónada `Maybe` y hemos visto como añade un contexto de que existan posibles fallos. Ya hemos aprendido a utilizar la mónada lista y hemos visto como nos permite introducir un no determinismo en nuestros programas. También hemos aprendido a trabajar con la mónada `IO`, ¡incluso antes de que supiéramos de la existencia de las mónadas!

En este capítulo, vamos a ver una cuantas mónadas más. Veremos como éstas pueden conseguir que nuestros programas sean más claros permitiéndonos manejar todo tipo de valores como si fueran monádicos. El hecho de ver unas cuantas mónadas más también reforzará nuestro conocimiento acerca de ellas.

Todas las mónadas que vamos a ver forman parte del paquete `mtl`. Un paquete de Haskell es una colección de módulos. El paquete `mtl` viene con la plataforma Haskell así que probablemente ya lo tengas instalado. Para comprobarlo, ejecuta `ghc-pkg list` en la línea de comandos. Así podrás ver todos los paquetes que tienes instalados y uno de ellos debe ser `mtl`, seguido de un número de versión.

¿Writer? No la conozco

Hemos cargado una pistola con la mónada `Maybe`, la mónada lista y la mónada`IO`. Ahora vamos a hacer sitio en la recámara para la mónada `Writer` y ver que pasa cuando la disparamos.

Mientras que `Maybe` sirve para valores con el contexto adicional de un posible fallo y las listas son para valores no deterministas, la mónada`Writer` sirve para valores que tienen una especie de registres como contexto. La mónada `Writer` nos permite realizar cómputos de forma que los valores del registro se combinan en un solo registro que será adjuntado al resultado final.

Por ejemplo, podríamos querer equipar algunos valores con unas cadenas que explicaran lo que esta sucediendo, probablemente para luego depurar el código. La siguiente función toma el número de bandidos de una banda y nos dice si es una gran banda o no. Una función muy simple:

```
isBigGang :: Int -> Bool
isBigGang x = x > 9
```

Ahora, en lugar de que nos devuelva solo `True` o `False`, queremos que nos devuelve también una cadena de registro que nos indique que ha hecho la función. Para ello solo tenemos que devolver una cadena junto al valor `Bool`:

```
isBigGang :: Int -> (Bool, String)
isBigGang x = (x > 9, "Compared gang size to 9.")
```

Así que ahora en vez de devolver una valor `Bool`, devuelve una tupla cuyo primer es el resultado original y el segundo es la cadena que acompaña al resultado. Ahora este resultado tiene añadido un cierto contexto. Vamos a probarla:

```
ghci> isBigGang 3
(False,"Compared gang size to 9.")
ghci> isBigGang 30
(True,"Compared gang size to 9.")
```

Hasta aquí todo bien. `isBigGang` toma un valor normal y devuelve un valor con un determinado contexto. Como ya sabemos, pasar a esta función un valor normal no causa ningún problema. Pero, ¿y si ya tenemos un valor que tiene adjuntado una cadena, como por ejemplo `(3, "Smallish gang.")`, y queremos pasarlo a `isBigGang`? Parece que una vez más nos topamos con la misma pregunta: si tenemos una función que toma un valor normal y devuelve un valor con un cierto contexto, ¿cómo extraemos el valor de ese contexto y se lo pasamos a la función?

Cuando estábamos explorando la mónada `Maybe` creamos la función`applyMaybe`, la cual tomaba un valor de tipo `Maybe a` y una función del tipo `a -> Maybe b` y pasa ese valor `Maybe a` a la función, incluso aunque la función toma una valor del tipo `a` y no `Maybe a`. Conseguíamos hacer esto teniendo en cuenta el contexto de los valores `Maybe`, el cual era el de los valores con un posible fallo. Dentro de la función`a -> Maybe b` éramos capaces de tratar ese valor con absoluta normalidad, ya que `applyMaybe` (que luego vino a ser `>>=`) se encargaba de comprobar si el valor era `Nothing` o un valor `Just`.

Del mismo modo, vamos a crear una función que tome un valor con un registro añadido, como por ejemplo `(a, String)`, y una función del tipo`a -> (b, String)` a la que pasaremos el valor inicial. La llamaremos`applyLog`. Como un valor del tipo `(a, String)` no lleva asociado ningún contexto de un posible fallo, sino únicamente un registro adicional,`applyLog` se encargará de que el registro de la variable original no se pierda concatenándolo con el registro del resultado de la función. Aquí tienes la implementación de `applyLog`:

```
applyLog :: (a,String) -> (a -> (b,String)) -> (b,String)
applyLog (x,log) f = let (y,newLog) = f x in (y,log ++ newLog)
```

Cuando tenemos un valor dentro de un contexto y queremos pasar dicho valor a una función, normalmente intentamos separar el valor real del contexto, luego intentamos aplicar la función sobre ese valor y para terminar volvemos a considerar el contexto. Con la mónada `Maybe`, primero comprobamos si el valor era del tipo `Just x` y si lo era, tomábamos el valor `x` y lo aplicábamos a la función. En este caso es fácil encontrar el valor real, ya que estamos trabajando con una dupla que contiene el valor y un registro. Primero tomamos el valor, que es `x` y le aplicamos la función `f`. Obtenemos una dupla de `(y, newLog)`, donde `y` es el nuevo resultado y `newLog` es el nuevo registro. Sin embargo, si devolviéramos esto como resultado, el registro antiguo no se incluiría en el resultado, así que devolvemos una dupla `(y,log ++ newLog)`. Utilizamos `++` para concatenar ambos registros.

Aquí tienes `applyLog` en acción:

```
ghci> (3, "Smallish gang.") `applyLog` isBigGang
(False,"Smallish gang.Compared gang size to 9")
ghci> (30, "A freaking platoon.") `applyLog` isBigGang
(True,"A freaking platoon.Compared gang size to 9")
```

El resultado es similar al anterior, solo que el número de bandidos en la banda va acompañado de un registro. Unos cuantos ejemplos más:

```
ghci> ("Tobin","Got outlaw name.") `applyLog` (\x -> (length x, "Applied length."))
(5,"Got outlaw name.Applied length.")
ghci> ("Bathcat","Got outlaw name.") `applyLog` (\x -> (length x, "Applied length"))
(7,"Got outlaw name.Applied length")
```

Fíjate en el interior de la función lambda, `x` es un cadena normal y no una tupla. Además `applyLog` se encarga de concatenar los registros.

Monoides al rescate

Nota

¡Asegurate de saber lo que son los *monoides* si quieres continuar!

Ahora mismo, `applyLog` toma valores del tipo `(a,String)`, pero, ¿existe alguno motivo especial por el que lo registros deban ser del tipo `String`? Utilizamos `++` para unir los registros, así que, ¿no debería aceptar cualquier tipo de listas, y no solo listas de caracteres? Pues sí, debería. Podemos cambiar su tipo a:

```
applyLog :: (a,[c]) -> (a -> (b,[c])) -> (b,[c])
```

Ahora, el registro es una lista. El tipo de valores que contiene la lista debe ser el mismo tipo de que tienen los elementos de la lista original, a la vez que deben ser iguales a los que devuelve la función. De otro modo, no podríamos utilizar `++` para unirlos.

¿Debería función con cadenas de bytes? No hay ninguna razón para que no funcionase. Sin embargo, el tipo que hemos utilizado solo acepta listas. Parece que tendremos que crear una `applyLog` solo para cadenas de bytes ¡Pero espera! Tanto las listas como los cadenas de bytes son monoides. Como tal, ambas poseen instancias de la clase de tipos `Monoid`, lo cual significa que ambas implementan la función `mappend`. Y tanto par las listas como para las cadenas de bytes, `mappend` sirve para unir. Mira:

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> B.pack [99,104,105] `mappend` B.pack [104,117,97,104,117,97]
Chunk "chi" (Chunk "huahua" Empty)
```

¡Genial! Ahora `applyLog` pede funcionar con cualquier monoide. Tenemos que cambiar la declaración de tipo para que lo refleje, y también la implementación ya que tenemos cambiar `++` por `mappend`:

```
applyLog :: (Monoid m) => (a,m) -> (a -> (b,m)) -> (b,m)
applyLog (x,log) f = let (y,newLog) = f x in (y,log `mappend` newLog)
```

Como el valor que acompaña al valor original ahora puede ser cualquier tipo de monoide, ya no tenemos que porque ver la dupla como una valor y un registro, sino como una valor y un monoide. Por ejemplo, podemos tener una tupla que tenga el nombre de un producto y su precio como valor monoidal. Simplemente tenemos que utilizar el `newtype Sum` para asegurarnos de que los precios se suman. Aquí tienes un ejemplo de una función que añade la bebida para cierto tipo de comida de cowboy:

```
import Data.Monoid

type Food = String
type Price = Sum Int

addDrink :: Food -> (Food,Price)
addDrink "beans" = ("milk", Sum 25)
addDrink "jerky" = ("whiskey", Sum 99)
addDrink _ = ("beer", Sum 30)
```

Utilizamos cadenas para representar las comidas y un `Int` dentro de un`newtype Sum` para mantener el precio total. Recuerda, cuando utilizamos`mappend` con `Sum` el resultado será la suma de ambos parámetros:

```
ghci> Sum 3 `mappend` Sum 9
Sum {getSum = 12}
```

La función `addDrink` es bastante simple. Si estamos comiendo alubias, devuelve `"milk"` junto `Sum 25`, es decir 25 centavos dentro de un`Sum`. Si estamos comiendo cecina bebemos whisky y si estamos comiendo cualquier otra cosa bebemos cerveza. Aplicar esta función a una comida no sería muy interesante, pero si utilizamos `applyLog` para pasar una comida junto a un precio a esta función la cosa se vuelve más interesante:

```
ghci> ("beans", Sum 10) `applyLog` addDrink
("milk",Sum {getSum = 35})
ghci> ("jerky", Sum 25) `applyLog` addDrink
("whiskey",Sum {getSum = 124})
ghci> ("dogmeat", Sum 5) `applyLog` addDrink
("beer",Sum {getSum = 35})
```

La leche cuesta `25` centavos, pero si comemos alubias que cuestan `10`centavos, acabaremos pagando `35` centavos. Ahora se ve claramente como el valor que acompañamos no tiene porque ser siempre un registro, puede ser cualquier tipo de monoide y como se unan ambos valores dependerá de ese monoide. Cuando utilizamos registros, se concatenan, cuando utilizamos números, se suman, etc.

Como el valor que devuelve `addDrink` es una dupla del tipo`(Food,Price)`, podemos pasar el resultado a `addDrink` de nuevo, de forma que el resultado nos diga que vamos a beber y cuanto nos a costado en total. Aquí tienes una muestra:

```
ghci> ("dogmeat", Sum 5) `applyLog` addDrink `applyLog` addDrink
("beer",Sum {getSum = 65})
```

Si añadimos una bebida a un poco de carne de perro obtendremos una cerveza y otros `30` centavos de más, `("beer", Sum 35)`. Si utilizamos`applyLog` para pasar este último valor a `addDrink`, obtenemos otra cerveza y el resultado final será `("beer", Sum 35)`.

El tipo `Writer`

Ahora que hemos visto que un valor junto a un monoide puede actuar como un valor monoidal, vamos a explorar la instancia de `Monad` para esos valores. El módulo `Contol.Monad.Writer` exporta el tipo `Writer w a` junto su instancia de `Monad` y algunas funciones útiles para trabajar con valores de este tipo.

Primero vamos a explorar el tipo en si mismo. Para adjuntar un monoide a un valor solo tenemos que ponerlos juntos en una dupla. El tipo `Writter w a`es solo un `newtype` de la dupla. Su definición es muy simple:

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

Gracias a que esta definido con `newtype` podemos crear una instancia de`Monad` que se comporte de forma diferente a la instancia de las tuplas normales. El parámetro de tipo `a` representa el tipo del valor mientras que el parámetro de tipo `w` representa el valor monádico que adjuntamos al valor.

Su instancia de `Monad` se define así:

```
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in Writer (y, v `mappend` v')
```

Antes de nada vamos a ver `>>=`. Su implementación es básicamente la misma que `applyLog`, solo que ahora la dupla está contenida en el `newtypeWriter`, así que tenemos que extraerla con ayuda de un patrón. Tomamos el valor `x` y le aplicamos la función `f`. Esto nos da un valor del tipo`Writer w a` que, con ayuda de una expresión `let, lo ajustamos a unpatrón. Llamamos ``y` al nuevo resultado y utilizamos `mappend` para combinar el monodie antiguo con el nuevo. Juntamos ambos valores en una dupla , luego dentro del constructor `Writer` y por fin este será el resultado final.

¿Qué pasa con `return`? Tiene que tomar un valor e introducirlo en el contexto mínimo por defecto que pueda albergar dicho valor como resultado ¿Cúal será ese contexto para los valores del tipo `Writer`? Tiene sentido que si queremos que el valor del monoide afecte tan poco como sea posible utilizar `mempty`. Utilizamos `mempty` como identadad para los valores monoidales, como `""`, `Sum 0`, cadenas de bytes vacías, etc. Siempre que utilicemos `mempty` junto a `mappend` y algún otro valor monoidal, el resultado será el valor monoidal. Así que si utilizamos `return` para crear un valor del tipo `Writer` y luego utilizamos `>>=` para pasárselo a una función, el valor monoidal resultante será igual al que devuelva la función. Vamos a utitlizar `return` con el número `3` unas cuantas veces, pero cada vez con un monoide distinto:

```
ghci> runWriter (return 3 :: Writer String Int)
(3,"")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
ghci> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

Como `Writer` no tiene una instancia de `Show`, tenemos que utilizar`runWriter` para convertir los valores de `Writer` en tuplas normales que puedan ser mostradas. Para `String`, el valor monoidal es la cadena vacía. Con `Sum`, es `0` porque si sumamos 0 a algo, el resultado será el mismo. Para `Product` la identidad es `1`.

La instancia de `Writer` no posee ninguna implementación de `fail`, así que si un ajuste de patrones falla dentro de un bloque `do` se llamará a la función `error`.

Utilizando la notación `do` junto a `Writer`

Ahora que tenemos una instancia de `Monad` podemos utilizar la notación`do` con valores `Writer`. Es útil para cuando tenemos varios valores del tipo `Writer` y queremos hacer cosas con ellas. Al igual que la demás mónadas, podemos tratar estos valores como valores normales dejando que se ocupen del contexto por nosotros. En este caso, toda los valores monoidales se unen con `mappend` y por lo tanto se reflejan en el resultado final. Aquí tiene un ejemplo de uso de la notación `do` con `Writer`:

```
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = Writer (x, ["Got number: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    return (a*b)
```

`logNumber` toma un número y crea un valor `Writer` a partir de él. Utilizamos una lista de cadenas como monoide de forma que adjuntamos una lista unitaria que dice que número hemos utilizado. `multWithLog` es un valor del tipo `Writer` que multiplica un `3` y un `5` y se asegura que los registros de ambos números aparezcan en el resultado final. Utilizamos`resultado` para

devolver `a*b` como resultado. Como `return` toma un valor y lo introduce en el contexto mínimo por defecto, podemos estar seguros de que no añadirá nada al registro. Esto es lo que vemos si lo ejecutamos:

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5"])
```

A veces solo queremos que cierto valor monoidal sea incluido llegado el momento. Para ello tenemos la función `tell` que forma parte de la clase de tipos `MonadWriter`. Para la instancia de `Writer`, toma un valor monoidal, como `["This is going on"], y crea un valor del tipo ``Writer`con resultado `()` y como valor monoidal adjunto el valor que le hayamos pasado. Cuando tenemos un resultado como `()` no lo ligamos a ninguna variable. Aquí tienes como se vería `multWithLog` con un reporte adicional:

```
multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["Gonna multiply these two"]
    return (a*b)
```

Es importante que `return (a*b)` esté en la última línea porque la última línea de una expresión `do` es el resultado final del bloque entero. Si hubiésemos puesto `tell` en la última línea, `()` hubiera sido el resultado final de esta expresión `do`. Hubiéramos perdido el resultado de la multiplicación, además que el tipo de la expresión hubiera sido`multWithLog :: Writer () Int`. Sin embargo, el registro hubira sido el mismo. Aquí lo tienes en acción:

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5","Gonna multiply these two"])
```

Añadiendo registros a los programas

El algoritmo de Euclides es un algoritmo que toma dos números y calcula su máximo común divisor. Es decir, el número más grande que puede dividir a ambos. Haskell ya posee la función `gcb`, que hace exactamente esto, pero vamos a implementarla de nuevo para añadirle un registro. Aquí esta el algoritmo normal:

```
gcd' :: Int -> Int -> Int
gcd' a b
    | b == 0    = a
    | otherwise = gcd' b (a `mod` b)
```

El algoritmo es muy sencillo. Primero, comprueba si el segundo número es 0. Si lo es, entonces el resultado es el primer número. Si no lo es, entonces el resultado es el máximo común divisor del segundo número y del resto de dividir el primer número por el segundo. Por ejemplo, si queremos saber el máximo común divisor de 8 y 3 simplemente tenemos que seguir el algoritmo. Como 3 no es 0, tenemos que encontrar el máximo común divisor de de 3 y 2 (si dividimos 8 por 3, el resto es 2). Luego, tenemos que encontrar el máximo común divisor de 3 y 2. 2 aún no es igual 0, así que tenemos 2 y 1. El segundo número aún no es 0 así que volvemos a aplicar el algoritmo para obtener 1 y 0, ya que dividir 2 por 1 nos da como resto 0. Finalmente, como el segundo número es 0, el resultado final es 1. Vamos a ver si Haskell opina lo mismo:

```
ghci> gcd' 8 3
1
```

Lo hace. Ahora, queremos adjuntar un contexto a este resultado, y el contexto será un valor monoidal a modo de registro. Como antes, utilizaremos una lista de cadenas como monoide. De este modo, el tipo de la nueva función `gcd'` será:

```
gcd' :: Int -> Int -> Writer [String] Int
```

Todo lo que nos queda por hacer es añadir a la función los valores del registro. Así será el código:

```
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        gcd' b (a `mod` b)
```

Esta función toma dos valores `Int` normales y devuelve un`Writer [String] Int`. Es decir, un `Int` que contiene un contexto de registro. En caso de que `b` sea `0`, en lugar de únicamente devolver`a` como resultado, utilizamos una expresión `do` para unir un valor del tipo `Writer` con el resultado. Primero utilizamos `tell` para indicar que hemos terminado luego utilizamos `return` para devolver `a` como resultado del bloque `do`. En lugar de utilizar esta expresión `do`podíamos haber utilizado simplemente:

```
Writer (a, ["Finished with " ++ show a])
```

Aún así la expresión `do` parece más legible. Luego tenemos el caso en el que `b` no es igual a `0`. En este caso, indicamos que vamos a utilizar`mod` para averiguar cual es el resto de dividir `a` por `b`. La segunda línea del bloque `do` simplemente llama de forma de recursiva a`gcd'`. Recuerda que `gcd'` al final devuelve un valor del tipo`Writer`, así que es perfectamente válido que `gcd' b (a `mod` b)` sea una línea de la expresión `do`.

Vamos a probar esta nueva versión de `gcd'`. Su resultado es del tipo`Writer [String] Int` así que debemos extraer la dupla de este `newtype`. Luego, el primer componente de la dupla será el resultado.

```
ghci> fst $ runWriter (gcd' 8 3)
1
```

¡Bien! Ahora, ¿qué pasa cono el registro? Como el registro es una lista de cadenas, vamos a utilizar `mapM_ putStrLn` par mostrar las cadenas por pantalla:

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcd' 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Finished with 1
```

Es genial como hemos sido capaces de cambiar el algoritmo original a uno que devuelva lo que está sucediendo simplemente cambiando los valores normales por valores monádicos y dejando que la implementación de `>>=` para `Writer` se encargue de los registros por nosotros. Podemos añadir este mecanismo de registro casi a cualquier función. Solo tenemos que remplazar los valores normales por valores del tipo `Writer` y cambiar la aplicación normal de funciones por `>>=` (o por expresiones `do` si vemos que es más legible).

Construcción de listas ineficiente

Cuando utilizamos la mónada `Writer` hay que tener cuidado con que monoide utilizar, ya que utilizar listas como monoides puede resultar en una ejecución muy lenta. Esto se debe al uso de `++` de `mappend`, añadir una lista al final de otra puede ser muy costoso si una lista es muy larga.

En la función `gcd'`, el registro es rápido porque la lista se acaba pareciendo a esto:

```
a ++ (b ++ (c ++ (d ++ (e ++ f))))
```

Las listas son estructuras de datos que se construyen de izquierda a derecha, y esto último es eficiente porque primero construimos la parte izquierda de la lista y solo después de construirla añadimos una lista más larga a la derecha. Pero si no tenemos cuidado al utilizar la mónada `Writer` podemos producir listas que se parezcan a:

```
((((a ++ b) ++ c) ++ d) ++ e) ++ f
```

Esta lista se asocia por la izquierda en vez de por la derecha. No es eficiente porque cada vez que queramos añadir la parte derecha a la parte izquierda tiene que construir la parte izquierda desde el principio.

La siguiente función funciona igual que `gdc'`, solo que registra las cadenas al revés. Primero produce el registro del procedimiento y luego añade el paso actual al final del registro.

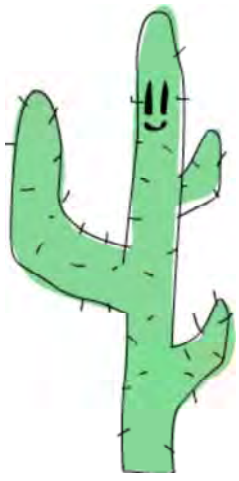
```
import Control.Monad.Writer

gcdReverse :: Int -> Int -> Writer [String] Int
gcdReverse a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        result <- gcdReverse b (a `mod` b)
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        return result
```

Primero realiza el paso de recursión y liga el resultado a `result`. Luego, añade el paso actual al registro, pero el paso actual debe ir al final del registro que a sido producido por la recursión. Al final, devuelve el resultado de la recursión como resultado final.

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcdReverse 8 3)
Finished with 1
2 mod 1 = 0
3 mod 2 = 1
8 mod 3 = 2
```

Es ineficiente porque acaba asociando el uso de `++` por la izquierda en lugar de por la derecha.

Listas de diferencia



Como la listas a veces son ineficientes cuando se concatenan repetidamente de esta forma, lo mejor es utilizar un estructura de datos que cuando se concatene sea siempre eficiente. Una estructura de este tipo es la lista de diferencia. Una lista de diferencia es similar a una lista, solo que en lugar de ser una lista normal, es una función que toma un lista y la antepone a otra lista. La lista de diferencia equivalente a la lista `[1,2,3]` sería la función `\xs -> [1,2,3] ++ xs`. Un lista vacía normal `[]` equivaldría a `\xs -> [] ++ xs`.

Lo interesante de las listas de diferencia es que soportan la concatenación de forma eficiente. Cuando añadimos los listas normales con `++`, hay que recorrer toda la lista de la izquierda hasta el final y luego añadir la otra ahí. Pero, ¿y si tomamos el enfoque de las listas de diferencia y representamos las listas como funciones? Bueno, entones añadir dos listas diferentes sería:

```
f `append` g = \xs -> f (g xs)
```

Recuerda que `f` y `g` son funciones que toman lista y la anteponen a otra lista. Así que, por ejemplo, si la función `f` es `("dog"++)` (que es otra forma de decir que es `\xs -> "dog" ++ xs`) y la función `g` es `("meat"++)`, entonces `f `append` g` crea una nueva función que será equivalente a:

```
\xs -> "dog" ++ ("meat" ++ xs)
```

Hemos concatenado dos listas de diferencia creando una nueva función que primero aplica una lista de diferencia y luego aplica la otra.

Vamos a crear un `newtype` para estas listas de diferencia de forma que podamos darle fácilmente una instancia de `Monoid`.

```
newtype DiffList a = DiffList { getDiffList :: [a] -> [a] }
```

El tipo que estamos definiendo es `[a] -> [a]` porque las listas de diferencia son solo funciones que toma una lista y devuelven otra. Convertir listas normales en listas de diferencia y viceversa es fácil:

```
toDiffList :: [a] -> DiffList a
toDiffList xs = DiffList (xs++)

fromDiffList :: DiffList a -> [a]
fromDiffList (DiffList f) = f []
```

Para crear una lista de diferencia a partir de una lista normal solo tenemos que hacer lo que ya hicimos antes, crear una función que añada una lista a ella. Como una lista de diferencia es una función que antepone algo a una lista, si queremos ese algo tenemos que aplicar la función a la lista vacía.

Aquí esta la instancia de `Monoid`:

```
instance Monoid (DiffList a) where
    mempty = DiffList (\xs -> [] ++ xs)
    (DiffList f) `mappend` (DiffList g) = DiffList (\xs -> f (g xs))
```

Fíjate que `mempty` es igual a `id` y `mappend` es en realidad una composición de funciones. Vamos a ver como funciona:

```
ghci> fromDiffList (toDiffList [1,2,3,4] `mappend` toDiffList [1,2,3])
[1,2,3,4,1,2,3]
```

Ahora podemos aumentar la eficiencia de la función `gcdReverse` haciendo que utilice listas de diferencia en lugar de listas normales:

```
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer (DiffList String) Int
gcd' a b
    | b == 0 = do
        tell (toDiffList ["Finished with " ++ show a])
        return a
    | otherwise = do
        result <- gcd' b (a `mod` b)
        tell (toDiffList [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)])
        return result
```

Solo tenemos que cambiar el tipo del monoide de `[String]` a `DiffListString` y luego cuando utilizamos `tell` convertir las listas normales a listas de diferencia con `toDiffList`. Vamos a ver si se parecen:

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ gcdReverse 110 34
Finished with 2
8 mod 2 = 0
34 mod 8 = 2
110 mod 34 = 8
```

Ejecutamos `gcdReverse 110 34`, luego utilizamos `runWriter` para extraer desde `newtype`, luego aplicamos `snd` para obtener el registro, y para terminar aplicamos `fromDiffList` para convertir la lista de diferencia en una lista normal que luego mostramos por pantalla.

Comparando el rendimiento

Para hacernos una idea de cuanto mejoran el rendimiento las listas de diferencia, considera esta función que simplemente hace una cuenta atrás hasta cero, pero produce el registro al revés, al igual que `gcdReverse`:

```
finalCountDown :: Int -> Writer (DiffList String) ()
finalCountDown 0 = do
    tell (toDiffList ["0"])
finalCountDown x = do
    finalCountDown (x-1)
    tell (toDiffList [show x])
```

Si le pasamos un `0`, lo registra. Para cualquier otro número, primero cuenta su predecesor y luego añade el número actual al registro. Así que si aplicamos `finalCountDown` a `100`, la cadena `"100"` será la última en registrar.

De cualquier modo, si cargamos esta función en *GHCi* y la aplicamos a un número muy grande, como `500000`, veremos que empieza a contar desde `0` rápidamente.

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ finalCountDown 500000
0
1
2
...
```

Sin embargo, si cambiamos la función para que utilice listas normales:

```
finalCountDown :: Int -> Writer [String] ()
finalCountDown 0 = do
    tell ["0"]
finalCountDown x = do
    finalCountDown (x-1)
    tell [show x]
```

Y luego le decimos a *GHCi* que empiece a contar:

```
ghci> mapM_ putStrLn . snd . runWriter $ finalCountDown 500000
```

Veremos que va muy despacio.

Por supuesto, esta no es la forma correcta y científica de probar lo rápidos que son nuestros programas, pero al menos podemos ver que para este caso, utilizar listas de diferencia produce resultados de la forma apropiada mientras que las listas normales tardan una eternidad.

Por cierto, te estará rondando por la cabeza el estribillo de la canción*Final Countdown* de *Europe*, así que, ¡disfrútala!

¿Reader? O no, otra vez la misma broma...

En el capítulo que hablábamos de los*funtores aplicativos*, vimos que el tipo función,`(->) r` posee una instancia de `Functor`. Al mapear una función `f`sobre una función `g` creamos una función que tomará los mismo parámetros que `g`, aplicará la función `g` y luego aplicará `f` a su resultado. Básicamente estamos creando una función igual que `g`, solo que en vez de devolver su resultado, devuelve el resultado de aplicar `f`. Por ejemplo:

```
ghci> let f = (*5)
ghci> let g = (+3)
ghci> (fmap f g) 8
55
```

También vimos que las funciones son funtores aplicativos. Nos permiten operar sobre funciones como si se tratasen de los resultados. Un ejemplo:

```
ghci> let f = (+) <$> (*2) <*> (+10)
ghci> f 3
19
```

La expresión `(+) <$> (*2) <*> (+10)` crea una función que toma un número, pasa ese número a `(*2)` y a `(+10)` y luego suma ambos resultados. Por ejemplo, si aplicamos esta función a `3`, aplica tanto `(*2)` como`(+10)` a `3`, obteniendo `6` y `13` y luego los suma devolviendo`19`.

El tipo función `(->) r` no es solo un funtor y un funtor aplicativo, sino también una mónada. De la misma forma que cualquier otro valor monádico que ya hemos visto, una función puede ser vista como un valor con un contexto. El contexto en este caso es que el valor aún no esta presente de forma que tenemos que aplicar esa función a algo antes de obtener el resultado.

Como ya sabemos como funcionan las funciones como funtores y funtores aplicativos, vamos a ver como luce su instancia de `Monad`. Se encuentra en`Control.Monad.Instances`.

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

Ya vimos como se implementaba `pure` para las funciones y `return` es básicamente lo mismo. Toma un valor y lo introduce en el contexto mínimo que siempre tendrá como resultado ese valor. Y la única forma de crear una función que siempre tenga el mismo resultado es ignorando completamente su parámetro.

La implementación de `>>=` puede parecer algo compleja, pero en realidad en muy sencilla. Cuando utilizamos `>>=` para pasar un valor monádico a una función, el resultado siempre es un valor monádico. Así que en este caso, pasamos una función a otra función, y resultado será también una función. Por este motivo la definición de `>>=` es una función lambda. Todas las implementaciones de `>>=` que hemos visto hasta ahora siempre asilaban el resultado del valor monádico de algún modo y luego le aplicaban la función`f`. Aquí pasa lo mismo. Para obtener el resultado de una función, tenemos que aplicarla a algo, por este motivo hacemos `(h w)` aquí, para obtener el resultado de una función y luego le aplicamos `f`. `f` devuelve un valor monádico, que es una función en este caso, así que que le aplicamos `w` de nuevo.

Si no entiendes como funciona `>>=` en este momento, no te preocupes, con unos cuantos ejemplos veremos que es una mónada muy simple. Aquí tienes un ejemplo de como usar una expresión `do` con esta mónada:

```
import Control.Monad.Instances

addStuff :: Int -> Int
addStuff = do
    a <- (*2)
```

```
    b <- (+10)
    return (a+b)
```

Es básicamente la misma expresión aplicativa que escribimos antes, solo que ahora vemos las funciones como mónadas. Una expresión `do` siempre resulta en un valor monádico. En este caso tomamos un número y luego aplicamos `(*2)` a ese número y el resultado lo ligamos a `a`. `(+10)` se aplica de nuevo al mismo número y ligamos su resultado a `b`. `return`, como en todas las demás mónadas, no tiene ningún otro efecto aparte de el de crear un valor monádico que contendrá algún resultado. En este caso crea una función que contendrá `(a+b)` como resultado. Si lo probamos veremos que obtenemos los mismos resultados:

```
ghci> addStuff 3
19
```

Tanto `(*2)` como `(+10)` se aplican al número `3`. `return (a+b)`también se aplica a `3` pero lo ignora y siempre devuelve `(a+b)` como resultado. Por este motivo, la mónada de las funciones es conocida como la mónada lectora (*reader* en inglés, en contraposición de *writer*, escritora). Todas las funciones leen de la misma fuente. Podemos ilustrar esto incluso mejor, podemos reescribir `addStuff` como:

```
addStuff :: Int -> Int
addStuff x = let
    a = (*2) x
    b = (+10) x
    in a+b
```

Podemos ver como la mónada lectora nos permite tratar a las funciones como valores en un cierto contexto. Podemos actuar como ya conociéramos lo que van a devolver. Lo que hacemos es unir todas las funciones en una sola y luego pasamos el parámetro de esta función a todas las demás. Si tenemos un montón de funciones a las que les faltan un solo parámetro y al final este parámetro será igual para todas, podemos utilizar la mónada lectora para extraer sus futuros resultados y la implementación de `>>=` se encargará de que todo funcione al final.

Mónadas monas con estado

Haskell es un lenguaje puro y como tal, los programas consisten en funciones que no pueden cambiar ningún estado global o variables, solo pueden hacer algunos cálculos o cómputos y devolver resultados. Esta restricción hace que sea más fácil razonar acerca de los programas ya que no tenemos que preocuparnos por el estado de una variable a lo largo del tiempo. Sin embargo, algunos problemas posee de forma inherentemente estados que cambian con el tiempo. Aunque estos estos estados no causan ningún problema a Haskell, a veces pueden ser un poco tediosos de modelar. Por esta razón Haskell posee la mónada estado, la cual nos permite tratar los problemas con estados como si fueran un juego de niños y además mantiene todo el código puro.

Cuando estábamos trabajando con *números aleatorios*, utilizábamos funciones que tomaban un generador de aleatoriedad como parámetro y devolvían un número aleatorio y un nuevo generador de aleatoriedad. Si queríamos generar varios números aleatorios, siempre teníamos que utilizar el generador de aleatoriedad que devolvió la función anterior. Si queremos crear una función que tome un generador de aleatoriedad y devuelva el resultado de lanzar una moneda tres veces, tenemos que hacer esto:

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
```

```
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

Toma un generador `gen` y luego `random gen` devuelve un `Bool` junto con un nuevo generador. Para lanzar la segunda moneda, utilizamos el nuevo generador, y así sucesivamente. La mayoría de los otros lenguajes no hubieran devuelto un nuevo generador junto con el número aleatorio. Simplemente habrían modificado el generador original. Pero Haskell es puro, no podemos hacer esto, así que tenemos que tomar un estado, crear un resultado a partir de él y producir un nuevo estado que será utilizado para generar nuevos resultados.

Si crees que para evitar tratar manualmente con estos estado en Haskell tenemos que perder la pureza de nuestro código, estás equivocado. Existe una pequeña mónada, llamada la mónada estado, que se encarga de manejar todo lo relaciona con estado sin renegar a la pureza.

Así que, para entender mejor todo este concepto de cómputos con estado vamos a darle un tipo. Antes hemos dicho que un cómputo con estado es una función que toma un estado y produce un resultado junto a un nuevo estado. Esta función tendría un tipo como este:

```
s -> (a,s)
```

`s` es el estado y `a` el resultado de estos cómputos con estado.

Nota
En otros lenguajes, la asignación de variables puede verse como un especie de cómputo con estado. Por ejemplo, si hacemos `x = 5` en un lenguaje imperativo, se asignará el valor `5` a la variable`x` y la expresión tendrá un resultado igual a `5`. Podemos ver esta funcionalidad como si la asignación fuera una función que toma un estado (es decir, todas las variables que han sido asignadas anteriormente) y devuelve un resultado (en este caso `5`) y nuevo estado que será el conjunto de todas las variables anteriores más la nueva asignación.

Estos cómputos con estado, funciones que toman un estado y devuelven un resultado junto con un nuevo estado, también se pueden ver como un valor en cierto contexto. El valor real es es el resultado, mientras que el contexto es el estado inicial del que hemos extraído el resultado, generando así un nuevo estado.

Pilas y pilones

Digamos que queremos modelar una pila. Tenemos un pila de cosas una encima de otra y podemos o bien añadir otra cosa encima de la pila o bien tomar una cosa de la cima de la pila. Cuando ponemos un objeto en la cima de la pila decimos que estamos apilando un objeto, y cuando tomamos un objeto de la pila decimos que estamos retirando un objeto. Si queremos el objeto que se encuentra más abajo de la pila tenemos que retirar antes todos los objetos que se encuentran por encima de éste.

Utilizaremos una lista para representar la pila, y su cabeza para representar la cima de la pila. Para hacer las cosas más fáciles, vamos a crear dos funciones: `pop` y `push`. `pop` tomará una pila y retirará un elemento que devolverá como resultado, junto a una nueva pila sin dicho elemento en la cima. `push` tomará un elemento y una pila y luego apilará dicho elemento en la pila. Devolverá `()` como resultado, junto a una nueva pila.

```
type Stack = [Int]

pop :: Stack -> (Int,Stack)
pop (x:xs) = (x,xs)

push :: Int -> Stack -> ((),Stack)
push a xs = ((),a:xs)
```

A la hora de apilar un elemento devolvemos `()` porque el hecho de apilar un elemento no tienen ningún resulto importante, su principal objetivo es modificar la pila. Fíjate que en `push` solo hemos añadido el primer parámetro, obteniendo así un cómputo con estado. `pop` ya es de por si un cómputo con estado debido a su tipo.

Vamos a escribir un trocito de código que simule el uso de estas funciones. Tomaremos una pila, apilaremos un `3` y luego retiraremos dos elementos, para pasar el rato más que nada.

```
stackManip :: Stack -> (Int, Stack)
stackManip stack = let
    ((),newStack1) = push 3 stack
    (a ,newStack2) = pop newStack1
    in pop newStack2
```

Tomamos una pila (`stack`) y luego hacemos `push 3 stack`, lo que nos devuelve una tupla. La primera parte de la tupla es `()` y la segunda es una nueva pila que llamaremos `newStack1`. Luego, retiramos un número de`newStack1`, lo cual devuelve ese número `a` (que es `3`) y una nueva pila que llamaremos `newStack2`. Luego retiramos otro elemento de`newStack2` y obtenemos un número `b` y una pila `newStack3`. Devolvemos una dupla que contendrá ese número y esa tupla. Vamos a probarlo:

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

Genial, el resultado es `5` y la pila es `[8,2,1]`. El mismo`stackManip` es un cómputo con estado. Hemos tomado un puñado de cómputos con estado y de alguna forma los hemos unido todos. Mmm... Me recuerda a algo.

El código que acabamos de ver es algo tedioso ya que tenemos que pasar el estado manualmente en cada cómputo, además de que tenemos que ligarlo a una variable para luego pasarlo al siguiente cómputo ¿No sería mejor si, en lugar de pasar una pila manualmente a cada función, pudiéramos escribir algo como esto?

```
stackManip = do
    push 3
    a <- pop
    pop
```

Bueno, pues usando la mónada estado podemos hacerlo. Gracias a ella podemos tomar cómputos con estado como estos y usarlos sin tener que preocuparnos por manejar el estado de forma manual.

La mónada estado

El módulo `Control.Monad.State` contiene un `newtype` para los cómputos con estado. Aquí tienes su definición:

```
newtype State s a = State { runState :: s -> (a,s) }
```

Un `State s a` es un cómputo con estado que manipula el estado del tipo`s` y tiene como resultado el tipo `a`.

Ahora que ya hemos visto como funcionan los cómputos con estado y que incluso podemos verlos como valores en cierto contexto, vamos a comprobar su instancia de `Monad`:

```
instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState
```

Vamos a echar un vistazo primero a `return`. Nuestro objetivo con `return`es tomar un valor y crear un cómputo con estado que siempre contenga ese valor como resultado. Por este motivo creamos una función lamba`\s -> (s,a)`. Siempre devolvemos `x` como resultado del cómputo con estado y además el estado se mantiene constante, ya que `return` debe insertar un valor en el contexto mínimo. Recapitulando, `return` tomará un valor y creará un cómputo con estado que revolverá ese valor como resultado y mantendrá el estado intacto.

¿Y `>>=`? Bueno, el resultado de pasar un cómputo con estado a una función con `>>=` es un cómputo con estado ¿no? Así que empezamos construyendo el`newtype State` y luego utilizamos una función lambda. La función lambda será el cómputo con estado. Pero, ¿qué es lo que hace? Bueno, de alguna forma debemos extraer el resultado del primer cómputo con estado. Como nos encontramos dentro de un cómputo con estado, podemos pasarle el estado actual`s` a `h`, lo cual devolverá un dupla con el resultado y un nuevo estado`(a, newState)`. Siempre que hemos implementado `>>=`, una vez extraído el resultado de un valor monádico aplicábamos la función `f` sobre éste para obtener un nuevo valor monádico. Por ejemplo, con `Writer`, luego de obtener el nuevo valor monádico, aún teníamos que asegurarnos de tratar el nuevo contexto aplicando `mappend` entre el valor monoidal antiguo y el nuevo. Aquí, realizamos `f a` para obtener un nuevo cómputo con estado `g`. Ahora que ya tenemos un nuevo cómputo con estado y nuevo estado (con el nombre de`newState`) solo tenemos que aplicar `g` sobre `newState`. El resultado será una tupla, y al mismo tiempo, el resultado final.

Así que `>>=` básicamente se encarga de unir dos cómputos con estado, solo que el segundo está oculto dentro de una función que se encarga de obtener el resultado anterior. Como `pop` y `push` son ya cómputos con estado, es muy fácil introducirlos dentro de `State`.

```
import Control.Monad.State

pop :: State Stack Int
pop = State $ \(x:xs) -> (x,xs)

push :: Int -> State Stack ()
push a = State $ \xs -> ((),a:xs)
```

`pop` ya es en si mismo un cómputo con estado y `push` es una función que toma un `Int` y devuelve un cómputo con estado. Ahora podemos reescribir el ejemplo anterior que apilaba un `3` y luego retiraba dos números así:

```
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

¿Ves como hemos unido un `push` y dos `pop` juntos en un solo cómputo con estado? Cuando extraemos el contenido del `newtype` obtenemos una función a la que tenemos que pasarle el estado inicial:

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

De hecho no tenemos porque ligar el segundo `pop` a `a` ya que no utilizamos `a` luego. Así que podemos reescribirlo de nuevo:

```
stackManip :: State Stack Int
stackManip = do
    push 3
    pop
    pop
```

Perfecto. Ahora queremos hacer esto: retiramos un número de la pila y si dicho número es `5` lo devolvemos a la pila, si no, apilamos un `3` y un `8`. Así sería el código:

```
stackStuff :: State Stack ()
stackStuff = do
    a <- pop
    if a == 5
        then push a
        else do
            push 3
            push 8
```

Bastante sencillo. Vamos a ejecutarlo junto a un estado inicial.

```
ghci> runState stackStuff [9,0,2,1,0]
((),[8,3,0,2,1,0])
```

Recuerda que las expresiones `do` devuelve valores monádicos y en el caso de la mónada `State`, cada expresión `do` es también una función con estado. Como tanto `stackStuff` y `stackManip` son cómputos con estado normales y corrientes, podemos unirlos y producir un nuevo cómputo con estado.

```
moreStack :: State Stack ()
moreStack = do
    a <- stackManip
    if a == 100
        then stackStuff
        else return ()
```

Si el resultado de `stackManip` sobre la pila actual es `100`, ejecutamos`stackStuff`, si no no hacemos nada. `return ()` simplemente mantiene el estado.

El módulo `Control.Monad.State` contiene una clase de tipos llamada`MonadState` y ésta a su vez contiene dos útiles funciones: `get` y`put`. Para `State`, `get` se implementa así:

```
get = State $ \s -> (s,s)
```

Es decir, toma el estado actual y lo devuelve como resultado. La función`put` toma un estado y crea una función con estado que remplazará el estado actual por su parámetro:

```
put newState = State $ \s -> ((),newState)
```

Gracias a estas funciones, podemos ver que el contenido de la pila actual o incluso remplazar toda la pila por una nueva.

```
stackyStack :: State Stack ()
stackyStack = do
    stackNow <- get
    if stackNow == [1,2,3]
        then put [8,3,1]
        else put [9,2,1]
```

Es bueno ver como quedaría el tipo de `>>=` si solo funcionará con valores del tipo `State`:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

Fíjate en que el tipo del estado `s` se mantiene constante pero sin embargo el tipo del resultado puede cambiar de `a` a `b`. Esto significa que podemos unir varios cómputos con estado cuyos resultados sean de diferentes tipos pero el tipo de sus estados sea el mismo. Y, ¿por qué? Bueno, por ejemplo, para `Maybe`, `>>=` tiene este tipo:

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

Tiene sentido que la mónada en si misma, `Maybe`, no cambie. No tendría sentido que pudiéramos usar `>>=` con dos mónadas distintas. Bueno, en el caso de la mónada estado, en realidad la mónada es `State s`, así que si`s` fuera distinta, estaríamos utilizando `>>=` entre dos mónadas distintas.

Aleatoriedad y la mónada estado

Al principio de esta sección vimos que como se generaban número aleatorios y que a veces puede ser algo pesado ya que cada función aleatoria toma un generador y devuelve un número aleatorio junto un nuevo generador, que tendremos que utilizar en lugar del viejo para generar otro número diferente. La mónada estado hace que trabajar con todo esto sea mucho más cómodo.

La función `random` del módulo `System.Random` tiene este tipo:

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

Es decir, toma un generador de aleatoriedad y produce un número aleatorio junto un nuevo generador. Podemos ver que en realidad se trata de un cómputo con estado, así que podemos introducirlo en el constructor `newtypeState` y luego utilizarlo como un valor monádico de forma que no nos tengamos que preocupar por manejar el estado:

```
import System.Random
import Control.Monad.State

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

Así que si ahora queremos lanzar tres monedas (`True` cruz, `False` cara) solo tenemos que hacer lo siguiente:

```
import System.Random
import Control.Monad.State

threeCoins :: State StdGen (Bool,Bool,Bool)
threeCoins = do
    a <- randomSt
    b <- randomSt
    c <- randomSt
    return (a,b,c)
```

Ahora `threeCoins` es un cómputo con estado y luego de tomar un generador de aleatoriedad inicial, lo pasa al primer `randomSt`, el cual producirá un número aleatorio y un nuevo generador, el cual será pasado al siguiente y así sucesivamente. Utilizamos `return (a,b,c)` para devolver `(a,b,c)` como resultado manteniendo constante el generador más reciente.

```
ghci> runState threeCoins (mkStdGen 33)
((True,False,True),680029187 2103410263)
```

Ahora realizar todo este tipo de tareas que requieren el uso de algún tipo de estado es mucho más cómodo.

Errores, errores, errores...

Sabemos que `Maybe` se utiliza para dar el contexto de un posible fallo a los valores. Un valor puede ser `Just algo` o `Nothing`. Sin embargo, cuando tenemos un `Nothing`, puede resultar útil dar alguna información del error que ha ocurrido, lo cual no podemos hacer con `Nothing`.

Por otra parte, el tipo `Either e a` nos permite incorporar el contexto de un posible fallo al mismo tiempo que nos permite dar información acerca del posible fallo, de forma que podemos describir que ha ido mal o dar alguna información acerca del fallo. Un valor del tipo `Either e a` puede ser un valor `Right`, lo cual representa un respuesta correcta, o un valor`Left`, que representa un fallo. Por ejemplo:

```
ghci> :t Right 4
Right 4 :: (Num t) => Either a t
ghci> :t Left "out of cheese error"
Left "out of cheese error" :: Either [Char] b
```

Básicamente es como un `Maybe` mejorado, así que tiene sentido que sea una mónada. También puede ser visto como un valor con el contexto de un posible fallo, solo que ahora existe un valor añadido cuando ocurre un error.

Su instancia de `Monad` es parecida a la de `Maybe` y se encuentra en `Control.Monad.Error`:

```
instance (Error e) => Monad (Either e) where
    return x = Right x
    Right x >>= f = f x
    Left err >>= f = Left err
    fail msg = Left (strMsg msg)
```

`return`, como siempre, toma un valor y lo introduce en el contexto mínimo por defecto. Introduce un valor en el constructor `Right` ya que utilizamos `Right` para representar un cómputo con éxito donde existe un resultado. Se parece mucho al `return` de `Maybe`.

La función `>>=` examina dos posibles casos: un `Left` y un `Right`. En caso de `Right`, la función `f` se aplica sobre el valor interior, de forma similar a lo que sucedía con `Just`. En caso de que ocurra un error, se mantiene constante el valor de `Left`, el cual da información acerca del error.

La instancia de `Monad` para `Either e` tiene un requerimiento adicional, y este es que el tipo del valor que está contenido en `Left`, el parámetro de tipo `e` en este caso, tiene que formar parte de la clase de tipos`Error`. La clase de tipos `Error` es para los tipos cuyos valores pueden actuar como mensajes de error. Define la función `strMsg`, que toma un error en forma de cadena y devuelve ese valor en forma de error. Un buen ejemplo de instancia de `Error` es el tipo `String`. Para el caso de `String`, la función `strMsg` simplemente devuelve la cadena que se le pasa:

```
ghci> :t strMsg
strMsg :: (Error a) => String -> a
ghci> strMsg "boom!" :: String
"boom!"
```

Como normalmente utilizamos `String` para describir los errores no tenemos que preocuparnos mucho por esto. Cuando un ajuste de patrones falla dentro de una expresión `do`, se devuelve valor `Left` para representar este error.

De cualquier modo, aquí tienes unos cuantos ejemplos:

```
ghci> Left "boom" >>= \x -> return (x+1)
Left "boom"
ghci> Right 100 >>= \x -> Left "no way!"
Left "no way!"
```

Cuando utilizamos `>>=` para pasar un valor `Left` a una función, la función se ignora y se devuelve un `Left` idéntico. Cuando pasamos un valor`Right` a una función, la función se aplica al contenido de éste, pero en este caso la función devuelve un valor `Left` de todas formas.

Si intentamos pasar una valor `Right``a una función que también devuelve unvalor ``Right` en *GHCi*, nos encontraremos con un error peculiar.

```
ghci> Right 3 >>= \x -> return (x + 100)

<interactive>:1:0:
    Ambiguous type variable `a' in the constraints:
      `Error a' arising from a use of `it' at <interactive>:1:0-33
      `Show a' arising from a use of `print' at <interactive>:1:0-33
    Probable fix: add a type signature that fixes these type variable(s)
```

Haskell dice que no sabe que tipo elegir para la parte `e` del tipo`Either e a`, incluso aunque solo mostremos la parte `Right`. Esto se debe a la restricción `Error e` de la instancia de `Monad`. Así que si no quieres ver más errores de este tipo cuando trabajes con la mónada `Either`, añade un anotación de tipo explícita:

```
ghci> Right 3 >>= \x -> return (x + 100) :: Either String Int
Right 103
```

¡Bien! Ahora funciona.

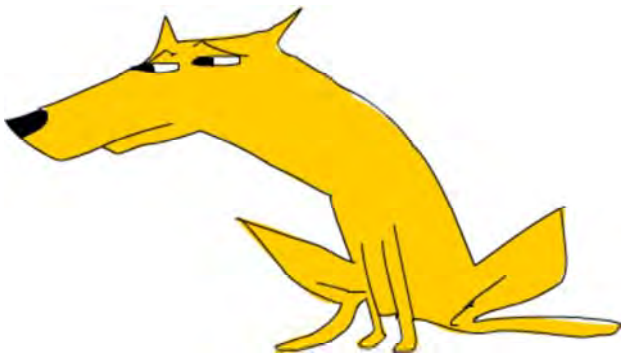
Aparte de este pequeño detalle, esta mónada es muy similar a la mónada`Maybe`. En el capítulo anterior, utilizamos varios aspectos monádicos de`Maybe` para simular el aterrizaje de las aves en la barra de equilibrio de nuestro buen amigo Pierre. A modo de ejercicio, puedes reescribir estas funciones con la mónada error de forma que cuando el funambulista se caiga, podamos informar del número de pájaros que había en la barra cuando se cayó.

Algunas funciones monádicas útiles

En esta sección vamos a ver una cuantas funciones que pueden operar con valores monádicos o devolver valores monádicos como resultado (¡o ambas cosas!). Normalmente no referimos a estas funciones como funciones monádicas. Mientras que algunas de éstas nos serán totalmente desconocidas, otras son las versiones monádicas de algunas funciones que ya conocemos, como `filter` o`foldl`.

`liftM` y sus amigos

Cuando empezamos nuestro viaje hacia la cima de las mónadas, primero vimos los funtores, que son cosas que se pueden mapear. Luego vimos que podíamos mejorar los funtores en algo que llamamos funtores aplicativos, que permitían aplicar funciones normales entre valores aplicativos a la vez que permitían tomar funciones e introducirlas en el contexto por defecto. Para terminar, vimos que podíamos mejorar los funtores aplicativos y eso nos llevaba a las mónadas, que añadían la habilidad de poder pasas esos valores con un cierto contexto a funciones normales.



Resumiendo, todas las mónadas son funtores aplicativos y todos los funtores aplicativos son funtores normales. La clase de tipos `Applicative` posee una restricción de clase que dice que su tipo debe poseer una instancia de la clase `Functor` antes de que se puede crear un instancia de `Applicative`. Aunque la clase `Monad` debería tener la misma restricción con`Applicative`,

ya que todas las mónadas son también funtores aplicativos, no la tiene. Esto se debe a que la clase de tipos `Monad` se introdujo en Haskell antes que `Applicative`.

Incluso aunque toda mónada es también un funtor, no tenemos que depender de la instancia de `Functor` gracias a la existencia de la función `liftM`.`liftM` toma una función y un valor monádico y mapea la función sobre el valor monádico. Vamos, ¡igual que `fmap`! Esta es su declaración de tipo:

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

Y esta es la de `map`:

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

Si tanto la instancia de `Functor` como la instancia de `Monad` obedecen las leyes de los funtores y de las mónadas, estas dos funciones hacen lo mismo (todas las mónadas que hemos visto cumplen ambas). Es lo mismo que pasaba con `pure` y `return`, solo que una tiene la restricción de clase`Applicative` y otra la de `Monad`. Vamos a probar `liftM`.

```
ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ Writer (True, "chickpeas")
(False,"chickpeas")
ghci> runWriter $ fmap not $ Writer (True, "chickpeas")
(False,"chickpeas")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
(101,[2,3,4])
```

Ya conocemos bastante bien como funciona `fmap` con los valores el tipo`Maybe`. `liftM` hace lo mismo. Para los valores del tipo `Writer`, la función se mapea sobre el primer componente de la dupla, que es el resultado. Hacer `fmap` o `liftM` sobre un cómputo con estado devuelve un nuevo cómputo con estado, solo que su resultado final se vera modificado por la función. Si no hubiésemos mapeado `(+100)` sobre `pop`, el resulto hubiese sido `(1,[2,3,4])`.

Esta es la implementación de `liftM`:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= (\x -> return (f x))
```

O con notación `do`:

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

Pasamos el valor monádico `m` a la función y luego aplicamos la función`f` a su resultado, introduciendo el resultado de ésta en el contexto por defecto. Gracias a las leyes de las mónadas, tenemos garantizado que el contexto se mantendrá constante, solo se modificará el resultado del valor monádico. Podemos ver que `liftM` está implementado sin hacer referencia a`Functor`. Esto significa que podemos implementar `fmap` (o `liftM`, depende de ti) utilizando únicamente lo que nos ofrecen las mónadas. Por este motivo, podemos concluir que las mónadas son más potentes que los funtores normales.

La clase de tipos `Applicative` nos permite aplicar funciones entre valores con un contexto como si se trataran de funciones normales.

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

Utilizar el estilo aplicativo hace las cosas muy fáciles. `<$>` es `fmap`y `<*>` es una función de la clase de tipos `Applicative` que tiene el siguiente tipo:

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

Es parecida a `fmap`, solo que la función en si misma posee un contexto. Tenemos que extraer de alguna forma el resultado de `f a` para poder mapear la función sobre ella y luego volver a introducir el resultado en un contexto. Como todas las funciones de Haskell están currificadas por defecto, podemos utilizar la combinación de `<$>` y `<*>` para aplicar una función sobre varios parámetros.

De cualquier forma, resulta que al igual que `fmap`, `<*>` también puede ser implementado utilizando lo que nos ofrece la clase de tipos `Monad`. La función `ap` es básicamente `<*>`, solo que posee un restricción de clase`Monad` en lugar de `Applicative`. Aquí tienes la definición:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf m = do
    f <- mf
    x <- m
    return (f x)
```

`mf` es un valor monádico cuyo resultado es una función. Como tanto la función como el valor están dentro de un contexto, extraemos la función del contexto y la llamamos `f`. Luego extraemos el valor y lo llamamos `x`. Para terminar aplicamos la función sobre el valor y devolvemos el resultado.

```
ghci> Just (+3) <*> Just 4
Just 7
ghci> Just (+3) `ap` Just 4
Just 7
ghci> [(+1),(+2),(+3)] <*> [10,11]
[11,12,12,13,13,14]
ghci> [(+1),(+2),(+3)] `ap` [10,11]
[11,12,12,13,13,14]
```

Ahora podemos ver que las mónadas son también más potentes que los funtores aplicativos, porque podemos utilizar las funciones de `Monad` para implementar las de `Applicative`. De hecho, a menudo la gente cuando sabe que un tipo es un mónada, primero implementa la instancia de `Monad` y luego crea la instancia de `Applicative` simplemente diciendo que `pure` es `return` y `<*>` es `ap`. De forma similar, si sabemos que algo tiene una instancia de `Monad`, podemos crear la instancia de `Functor` simplemente estableciendo que `fmap` es igual a `liftM`.

La función `liftA2` es una función de conveniencia para aplicar una función entre dos valores aplicativos. Su definición es así de sencilla:

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

La función `liftM2` hace exactamente lo mismo, solo que posee una restricción de clase `Monad`. También existe `liftM3`, `liftM4` y`liftM5`.

Hemos visto como las mónadas son más potentes que los funtores aplicativos y que los funtores normales y aunque todas las mónadas son también funtores y funtores aplicativos, no necesariamente poseen una instancia de `Functor`y `Applicative`, y esta es la razón por la que acabamos de ver las funciones equivalentes entre los funtores y las mónadas.

La función `join`

Piensa en esto: si el resultado de un valor monádico es otro valor monádico, es decir, si un valor monádico es anidado dentro de otro, ¿Podemos convertir ambos en un único valor monádico? Por ejemplo, si tenemos `Just (Just 9)`, ¿Podemos convertirlo en `Just 9`? Pues resulta que convertir valores monádicos anidados en valores monádicos simples es una de las propiedades únicas de las mónadas. Por este motivo tiene su razón de ser la función`join`.

```
join :: (Monad m) => m (m a) -> m a
```

Toma una un valor monádico que contiene otro valor monádico y devuelve un solo valor monádico. Aquí tienes un ejemplo de su uso con valores `Maybe`:

```
ghci> join (Just (Just 9))
Just 9
ghci> join (Just Nothing)
Nothing
ghci> join Nothing
Nothing
```

La primera línea tiene un cómputo correcto como resultado de otro cómputo correcto, así que ambos son unido en un solo cómputo correcto. La segunda línea posee un `Nothing` como resultado de un valor `Just`. Antes, cuando trabajamos con valores `Maybe` queríamos combinar varios valores en uno, ya sea con `<*>` o con `>>=`, todos tenían que ser valores `Just` para que el resultado fuese también un valor `Just`. Si existe un fallo en algún punto del camino, el resultado final será un fallo y lo mismo ocurre aquí. En la tercera línea, vemos que si intentamos unir un único fallo, el resultado es también un fallo.

Unir, o aplanar listas es bastante intuitivo:

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

Como puedes ver, para listas `join` es igual que `concat`. Para unir un valor `Writer` cuyo resultado es también un valor `Writer` tenemos que aplicar `mappend` al valor monádico.

```
ghci> runWriter $ join (Writer (Writer (1,"aaa"),"bbb"))
(1,"bbbaaa")
```

El valor monádico exterior `"bbb"` se utiliza primero y luego se le añade`"aaa"`. Dicho de otra forma, cuando queremos examinar el resultado de un valor `Writer`, primero tenemos que actualizar su registro y solo después de esto podremos examinar sus contenidos.

Unir valores `Either` es muy parecido a unir valores `Maybe`:

```
ghci> join (Right (Right 9)) :: Either String Int
Right 9
ghci> join (Right (Left "error")) :: Either String Int
Left "error"
```

```
ghci> join (Left "error") :: Either String Int
Left "error"
```

Si aplicamos `join` a un cómputo cuyo resultado sea otro cómputo con estado, el resultado será un cómputo con estado que primero ejecutará el cómputo exterior y luego el interior. Mira:

```
ghci> runState (join (State $ \s -> (push 10,1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])
```

Aquí la función lambda toma un estado y apila `2` y `1` sobre la pila y devuelve como resultado `push 10`, que es otro computo con estado. Así que cuando todo esto se une con `join` y luego se ejecuta, primero se apila`2` y `1` y luego se ejecuta `push 10`, poniendo así `10` en la cima de la pila.

La implementación de `join` es la siguiente:

```
join :: (Monad m) => m (m a) -> m a
join mm = do
    m <- mm
    m
```

Como el resultado de `mm` es un valor monádico, obtenemos ese resultado y luego simplemente lo ponemos solo en la última línea porque por si solo ya es un valor monádico. El truco aquí esta en `m <- mm`, el contexto de la mónada de la que estamos obteniendo un resultado se tiene en cuenta. Por este motivo, por ejemplo, los valores `Maybe` solo devuelven `Just` cuando tanto el valor exterior como el valor interior son ambos `Just`. Así se vería esto si `mm` fuera desde el principio `Just (Just 8)`:

```
joinedMaybes :: Maybe Int
joinedMaybes = do
    m <- Just (Just 8)
    m
```

Quizá lo más interesante de `join` es que funciona para cualquier mónada, pasar un valor monádico a una función con `>>=` es lo mismo que mapear esa función sobre el valor monádico y luego utilizar `join` para unir el resultado. Dicho de otro modo, `m >>= f` es siempre igual que`join (fmap f m)`. Vale la pena que le dediques un poco de atención. Con`>>=`, siempre pensamos que estamos pasando un valor monádico a una función que toma un valor normal pero devuelve un valor monádico. Si mapeamos directamente la función sobre el valor monádico, tendremos un valor como resultado un valor monádico dentro de otro valor monádico. Por ejemplo, digamos que tenemos `Just 9` y la función `\x -> Just (x+1)`. Si mapeamos esta función sobre `Just 9` nos dará como resultado `Just (Just 10)`.

El hecho de que `m >>= f` sea siempre igual a `join (fmap f m)` es muy útil porque si estamos creando una instancia de `Monad` para un tipo, siempre es más fácil averiguar como se deben unir dos valores monádicos anidados que averiguar como implementar `>>=`.

filterM

La función `filter` junto a la función `map` son de las funciones más utilizadas en Haskell. Toma un predicado y una lista y la filtra de forma que la lista resultante solo contenga los resultados que satisfagan el predicado.

```
filter :: (a -> Bool) -> [a] -> [a]
```

El predicado toma un elemento de la lista y devuelve un valor `Bool`. Pero, ¿y si el valor `Bool` que devuelve el predicado fuera en realidad un valor monádico? ¿¡Qué!? En otras palabras, ¿y si el resultado tuviera un contexto? ¿Podría funcionar? Por ejemplo, ¿qué pasaría ti todos los valores `True` y`False` que el predicado produce vienen acompañados de un valor monádico como `["Número 5 aceptado"]` o `["3 es muy pequeño"]`? Dicho así podría funcionar. Si ese fuera el caso, cabe esperar que la lista resultante venga con un registro que contenga el registro de todos los valores que se han ido produciendo. Así que si el valor `Bool` que produce el resultado viene con un contexto, lo normal es que la lista resultante también venga con un contexto, de otro modo el contexto de cada `Bool` se perdería.

La función `filterM` de `Control.Monad` hace exactamente lo que estamos buscando.

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

El predicado devuelve un valor monádico cuyo resultado es un `Bool`, pero como es un valor monádico, su contexto puede ser cualquier cosa, desde un fallo hasta un no determinismo. Para asegurarnos de que el resultado final refleja el contexto, el resultado también debe ser un valor monádico.

Vamos a tomar una lista y vamos a filtrarla de forma que solo nos quedemos con los números que sean menores que 4. Para empezar, vamos a utilizar la función normal `filter`:

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

Muy fácil. Ahora, vamos hacer que este predicado, además de devolver`True` o `False`, también adjunte un registro indicando lo que ha hecho. Por supuesto vamos a utilizar la mónada `Writer`.

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
    | x < 4 = do
        tell ["Keeping " ++ show x]
        return True
    | otherwise = do
        tell [show x ++ " is too large, throwing it away"]
        return False
```

En lugar de devolver un `Bool`, esta función devuelve un`Writer [String] Bool`. Es un predicado monádico. Suena excesivo, ¿no crees? Si el número es menor que `4` registramos que lo vamos a mantener y luego hacemos `return True`.

Ahora vamos a utilizar `filterM` con una lista. Como el predicado devuelve un valor `Writer`, el resultado de la lista será también un valor`Writer`.

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

Examinando el resultado del valor de tipo `Writer` vemos que todo está en orden. Ahora, vamos a mostrar el registro:

```
ghci> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 is too large, throwing it away
Keeping 1
5 is too large, throwing it away
Keeping 2
10 is too large, throwing it away
Keeping 3
```

Increíble. Simplemente utilizando un predicado monádico con `filterM` somos capaces de filtrar una lista a la vez que mantenemos el contexto del que estamos utilizando.

Existe un truco en Haskell para obtener el superconjunto de una lista (si vemos las listas como un conjunto). El superconjunto de un conjunto es un conjunto de todos los posible subconjuntos de éste. Así que si tenemos un conjunto como `[1,2,3]`, su superconjunto incluirá los siguientes conjuntos:

```
[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
```

En otras palabras, obtener el superconjunto es como obtener todas las posibles combinaciones de mantener o eliminar elementos de un conjunto. `[2,3]` sería el conjunto original al que hemos eliminado el número `1`.

Para crear una función que devuelva el superconjunto de una lista vamos a utilizar el no determinismo. Tomamos una lista como `[1,2,3]` y luego miramos el primer elemento, que es `1`, y nos preguntamos: ¿lo debemos mantener o lo debemos eliminar? Bueno, en realidad queremos hacer ambas cosas. Resumiendo, vamos a filtrar una lista y vamos a un utilizar un predicado no determinista que elimine y mantenga cada elemento de la lista.

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

¿Qué es esto? Bueno, elegimos eliminar y mantener cada elemento, independientemente del valor de dicho elemento. Tenemos un predicado no determinista, así que el resultado también será no determinista y por lo tanto su tipo será una lista de listas. Vamos a probarlo.

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Quizá esto puede que no se entienda a la primera, pero si consideramos las listas como valores no deterministas que no saben que valor escoger y por tanto deciden ser todos a la vez, es más fácil de ver.

foldM

La versión monádica de `foldl` es `foldM`. Si recuerdas bien los*pliegues*, sabrás que `foldl` toma una función binaria, un acumulador inicial y una lista y pliega la lista desde la izquierda reduciendo la lista a un solo valor. `foldM` hace básicamente lo mismo, solo que toma una función binaria que devuelve un valor monádico. Sorprendentemente, el resultado final también es un valor monádico. La declaración de tipo de `foldl` es:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Mientras que la de `foldM` es:

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

El valor que devuelve la función binaria es un valor monádico por lo tanto el valor final del pliegue también lo es. Vamos a sumar una lista de números con un pliegue:

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

El acumulador inicial es `0` y luego se suma `2` al acumulador, el resultado pasa a ser el nuevo acumulador que tiene un valor de `2`. Luego se suma `8` al acumulador devolviendo así `10` que pasa a ser el nuevo acumulador y así hasta que alcance el final de la lista, donde el acumulador final será el resultado final de la función.

¿Y si queremos sumar una lista de números y además queremos añadir la condición de que si en la lista existe un número mayor a `9`, todo el cómputo falle? Tendría sentido utilizar la función binaria para comprobar si el número actual es mayor que `9` y si lo es, falle, y si no continúe. Debido a esta nueva posibilidad de fallo, vamos a hacer que la función binaria devuelva un acumulador dentro de un tipo `Maybe` en lugar de un acumulador normal. Así sería la función binaria:

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
    | x > 9     = Nothing
    | otherwise = Just (acc + x)
```

Como la función binaria es ahora una función monádica, ya no podemos utilizar un pliegue normal como `foldl`, tendremos que usar un pliegue monádico.

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

¡Genial! Como había un número mayor que `9`, el resultado final fue`Nothing`. También es útil realizar un pliegue con una función binaria que devuelva un valor `Writer`, ya que de este modo podemos obtener un registro conforme recorremos la lista.

Creando una versión segura de la calculadora RPN

Cuando estábamos solucionando el problema de implementar la*calculadora RPN*, vimos que funcionaría bien siempre y cuando la entrada tuviera sentido. Pero si algo iba mal, el programa entero terminaba. Ahora que ya sabemos como convertir código en su versión monádica, vamos a hacer una versión de la calculador RPN más segura ayudándonos de la mónada`Maybe`.

Implementamos la calculadora RPN de forma que tomaba una cadena, como`"1 3 + 2 *"`, la dividiera en palabras para obtener algo como `["1","3","+","2","*"]` y luego la plegara utilizando como acumulador inicial una pila vacía y una función binaria que apilaba números en la pila, o sumaba los dos elementos superiores, o los dividía, etc.

Este era la función principal:

```
import Data.List

solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
```

Convertíamos la expresión en una lista de cadenas, la plegábamos utilizando una función binaria y luego devolvíamos el único elemento que quedaba en la pila. Así era la función binaria:

```
foldingFunction :: [Double] -> String -> [Double]
foldingFunction (x:y:ys) "*" = (x * y):ys
```

```
foldingFunction (x:y:ys) "+" = (x + y):ys
foldingFunction (x:y:ys) "-" = (y - x):ys
foldingFunction xs numberString = read numberString:xs
```

En este caso el acumulador del pliegue era la pila, la cual representábamos como una lista de valores `Double`. Conforme la función de pliegue avanzaba por la expresión RPN, si el elemento actual era un operador, obteníamos los dos elementos superiores de la pila, aplicábamos el operador y luego apilábamos el resultado de nuevo. Si el elemento actual era una cadena que representaba un número, convertíamos la cadena en un número real y lo apilábamos.

Primero vamos a hacer que esta función pueda fallar de forma correcta. Su declaración de tipo cambiará de esta forma:

```
foldingFunction :: [Double] -> String -> Maybe [Double]
```

Así que ahora su resultado será o bien un valor `Just` con una pila o bien fallará con `Nothing`.

La función `reads` es igual que `read`, solo que devuelve una lista con un único elemento en caso de que tenga éxito. Si no puede leer algo, devuelve una lista vacía. Aparte de devolver el valor que lee, también devuelve la parte de la cadena que no ha consumido. Vamos a decir que siempre tiene que leer toda la cadena para que funcione correctamente y vamos a crear una función `readMaybe` por conveniencia.

```
readMaybe :: (Read a) => String -> Maybe a
readMaybe st = case reads st of [(x,"")] -> Just x
                                _ -> Nothing
```

La probamos:

```
ghci> readMaybe "1" :: Maybe Int
Just 1
ghci> readMaybe "GO TO HELL" :: Maybe Int
Nothing
```

Vale, parece que funciona. Ahora vamos a convertir la función binaria en una función binaria que puede fallar.

```
foldingFunction :: [Double] -> String -> Maybe [Double]
foldingFunction (x:y:ys) "*" = return ((x * y):ys)
foldingFunction (x:y:ys) "+" = return ((x + y):ys)
foldingFunction (x:y:ys) "-" = return ((y - x):ys)
foldingFunction xs numberString = liftM (:xs) (readMaybe numberString)
foldingFunction _ _ = fail "¡whops!"
```

Los primeros tres casos son iguales que los originales, solo que ahora la pila nueva se introduce en `Just` (hemos utilizado `return` para lograrlo, pero podíamos haber utilizado simplemente `Just` también). En el cuarto caso, hacemos `readMaybe numberString` y luego mapeamos `(:xs)` sobre su resultado. Es decir, si la pila `xs` es `[1.0,2.0]` y`readMaybe numberString` devuelve `Just 3.0`, el resultado será`[1.0,2.0,3.0]`. Si `readMaybe numberString` devuelve `Nothing` el resultado final será `Nothing`. Vamos a probar esta función:

```
ghci> foldingFunction [3,2] "*"
Just [6.0]
ghci> foldingFunction [3,2] "-"
Just [-1.0]
ghci> foldingFunction [] "*"
Nothing
ghci> foldingFunction [] "1"
Just [1.0]
ghci> foldingFunction [] "1 wawawawa"
Nothing
```

¡Parece que funciona! Ahora es hora de mejorar la función `solveRPN` ¡Aquí la tienen!

```
import Data.List

solveRPN :: String -> Maybe Double
solveRPN st = do
    [result] <- foldM foldingFunction [] (words st)
    return result
```

Al igual que antes, tomamos una cadena y la dividimos en palabras. Luego, realizamos el pliegue, empezando con una pila vacía, solo que en lugar de realizar un pliegue normal con `foldl` utilizamos `foldM`. El resultado de `foldM` debe ser un valor del tipo `Maybe` que contendrá una lista (es decir, la pila final) que a su vez solo debería contener un valor. Utilizamos una expresión `do` para obtener el valor y lo llamamos`result`. En caso de que `foldM` devuelva `Nothing`, el resultado final será `Nothing`, porque así es como funciona la mónada `Maybe`. Fíjate también en el ajuste del patrón en el interior de la expresión `do`, de esta forma si la lista tiene más de un solo o ningún elemento, el ajuste fallará y se producirá un `Nothing`. En la última línea simplemente hacemos`return result` para devolver el resultado de la expresión RPN dentro de un valor del tipo `Maybe`.

Probémoslo:

```
ghci> solveRPN "1 2 * 4 +"
Just 6.0
ghci> solveRPN "1 2 * 4 + 5 *"
Just 30.0
ghci> solveRPN "1 2 * 4"
Nothing
ghci> solveRPN "1 8 wharglbllargh"
Nothing
```

El primer fallo sucede porque la pila final no contiene un único elemento y por tanto el ajuste de patrones contenido en la expresión `do` falla. El segundo fallo se debe a que `readMaybe` devuelve `Nothing`.

Componiendo funciones monádicas

Cuando hablamos de las leyes de las mónadas, vimos que la función `<=<` era parecida a la composición de funciones, solo que en lugar de tratar con funciones normales `a -> b`, funcionaba con funciones monádicas como`a -> m b`. Por ejemplo:

```
ghci> let f = (+1) . (*100)
ghci> f 4
401
ghci> let g = (\x -> return (x+1)) <=< (\x -> return (x*100))
ghci> Just 4 >>= g
Just 401
```

En este ejemplo primero componemos dos funciones normales, y luego las aplicamos la función resultante a `4`. Luego componemos dos funciones monádicas, y luego le pasamos `Just 4` a la función resultante utilizando`>>=`.

Si tenemos una lista de funciones, podemos componerlas en una sola gran función utilizando `id` como acumulador inicial y la función `.` como función binaria. O también utilizando la función `foldr1`. Aquí tienes un ejemplo:

```
ghci> let f = foldr (.) id [(+1),(*100),(+1)]
ghci> let g = foldr1 (.) [(+1),(*100),(+1)]
ghci> f 1
201
ghci> g 1
201
```

La función `f` toma un número y luego le suma `1`, luego multiplica el resultado por `100` y luego le suma `1` al resultado anterior. De todos modos, podemos componer funciones monádicas de la misma forma, solo que en lugar de utilizar una composición de funciones normal utilizamos `<=<`, y en lugar de utilizar `id` utilizamos `return`. No tenemos que utilizar`foldM` en lugar de `foldr` ya que `<=<` se encarga de que la composición sea monádica.

Cuando vimos la mónada lista en el *capítulo anterior*, la utilizamos para encontrar a qué posiciones podía desplazarse un caballo en un tablero de ajedrez con exactamente tres movimientos. Teníamos una función que se llamaba `moveKnight`, la cual tomaba la posición en el tablero del caballo y devolvía todos los posibles movimientos que podía tomar. Luego, para generar todos los posibles posiciones que podía alcanzar en tres movimientos utilizábamos una función como estas:

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

Y para comprobar si el caballo podía llegar desde `start` hasta `end` en tres movimientos utilizábamos:

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

Utilizando la composición de funciones podemos crear una función como `in3`, solo que en lugar de generar todas las posibles soluciones que puede alcanzar el caballo en tres movimientos, podemos hacerlo para un número arbitrario de movimientos. Si nos fijamos en `in3`, vemos que hemos utilizado`moveKnight` tres veces y hemos utilizado `>>=` en cada paso para pasar las posibles posiciones anteriores. Ahora vamos a hacerlo más general.

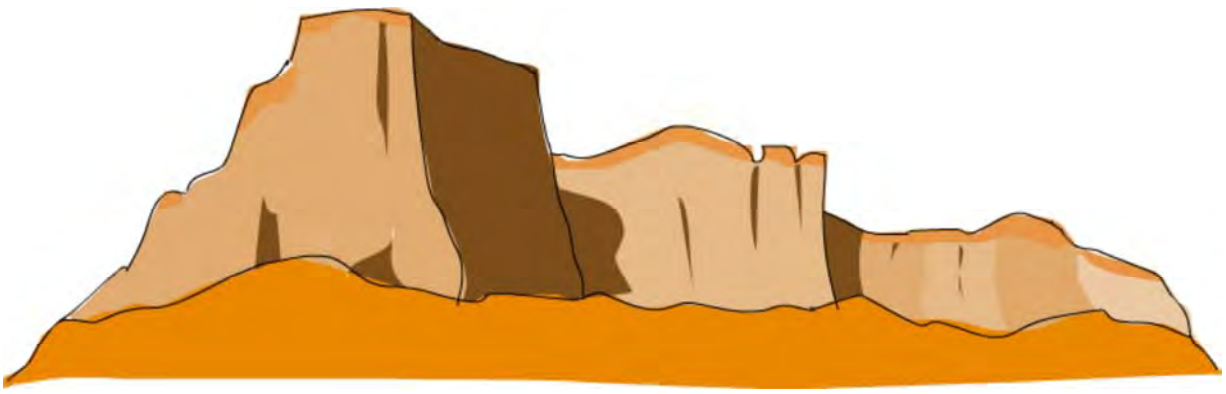
```
import Data.List

inMany :: Int -> KnightPos -> [KnightPos]
inMany x start = return start >>= foldr (<=<) return (replicate x moveKnight)
```

Primero utilizamos `replicate` para crear una lista que contenga `x` veces la función `moveKnight`. Luego, componemos monádicamente todas esas funciones en una, lo cual resulta en una función que toma una posición incial y mueve el caballo de forma no determinista `x` veces. Luego, simplemente creamos una lista unitaria con la posición inicial con `return` y se la pasamos a la función.

Ahora también podemos cambiar la función `canReachIn3` para que sea más general:

```
canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn x start end = end `elem` inMany x start
```

Creando mónadas



En esta sección vamos a ver un ejemplo de como se crea un tipo, como se identifica que se trata de una mónada y luego como darle una instancia`Monad` apropiada. Normalmente no nos ponemos a crear una mónada por el simple hecho de crear

mónadas. En lugar de ello, solemos crear un tipo con el propósito de modelar un aspecto de algún problema y luego si vemos que ese tipo representa valores con un contexto y puede comportarse como una mónada, le damos una instancia de `Monad`.

Como ya hemos visto, las listas se utilizan para representar valores no deterministas. Una lista como `[3,5,9]` puede ser vista como un solo valor no determinista que no puede decidir que valor ser. Cuando pasamos una lista a una función con `>>=`, simplemente crea todas las posibilidades de tomar un elemento concreto de la lista y le aplica la función, de forma que los resultados que produzca aparezcan en el resultado final.

Si vemos la lista `[3,5,9]` como los número `3`, `5` y `9` al mismo tiempo, podemos darnos cuenta de que no tenemos ninguna información de la probabilidad de que esos números aparezcan. ¿y si quisiéramos un modelar un valor no determinista como `[3,5,9]`, pero que expresará también que `3`tiene un 50% probabilidades de ocurrir y `5` y `9` tienen un 25% de probabilidades? Es una pregunta muy larga, lo se, pero vamos a buscar una respuesta.

Digamos que cada elemento de la lista va acompañado de otro valor, la probabilidad de que ocurra. Podría tener sentido representarlo así:

```
[(3,0.5),(5,0.25),(9,0.25)]
```

En las matemáticas, las probabilidades no se suelen representar con porcentajes sino con valores reales que van desde el 0 hasta el 1. Un 0 significa que no hay ninguna posibilidad de que ocurra un suceso mientras que un 1 representa que el suceso va ocurrir sí o sí. Los número en coma flotante pueden ser muy eficientes ya que tienden a perder precisión, así que Haskell nos ofrece un tipo de dato para los números racionales de forma que no pierda precisión. Este tipo se llama `Rational` y reside en el módulo`Data.Ratio`. Para crear un número del tipo `Rational` lo escribimos en forma de fracción. Separamos el numerador y el denominador por `%`. Aquí tienes unos ejemplos:

```
ghci> 1%4
1 % 4
ghci> 1%2 + 1%2
1 % 1
ghci> 1%3 + 5%4
19 % 12
```

La primera línea representa un cuarto. En la segunda línea sumamos dos medios y obtenemos la unidad y en la tercera línea sumamos un tercero a cinco cuartos y obtenemos diecinueve docenas de huevos. Vamos a utilizar número `Rational` en lugar de números en coma flotante para representar las probabilidades.

```
ghci> [(3,1%2),(5,1%4),(9,1%4)]
[(3,1 % 2),(5,1 % 4),(9,1 % 4)]
```

Vale, `3` tiene la mitad de posibilidades de ocurrir y `5` y `9` tienen un cuarto de posibilidades de salir. Muy bonito.

Tomamos las listas y les añadimos un contexto adicional, así que esto también representa valores en cierto contexto. Antes de continuar, vamos representar esto con `newtype` porque algo me dice que vamos a crear alguna instancias.

```
import Data.Ratio

newtype Prob a = Prob { getProb :: [(a,Rational)] } deriving Show
```

Muy bien ¿Esto es un funtor? Bueno, la lista es un funtor, así que probablemente esto sea un funtor también ya que solo hemos añadido algo más de contexto. Cuando mapeamos una función sobre una lista, la aplicamos a todos los elementos. Ahora también aplicaremos la función a todos los elementos, solo que mantendremos las probabilidades intactas.

```
instance Functor Prob where
    fmap f (Prob xs) = Prob $ map (\(x,p) -> (f x,p)) xs
```

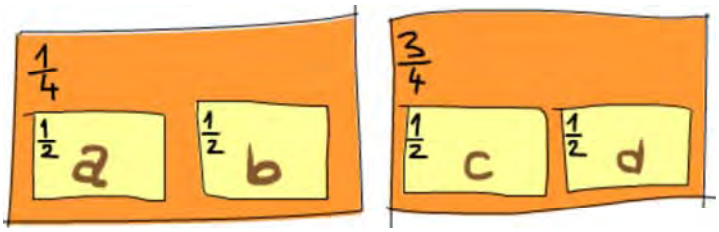
Extraemos el valor del `newtype` utilizando un ajuste de patrones, aplicamos la función `f` a los valores mientras nos aseguramos de mantener constantes las probabilidades. Vamos a ver si funciona:

```
ghci> fmap negate (Prob [(3,1%2),(5,1%4),(9,1%4)])
Prob {getProb = [(-3,1 % 2),(-5,1 % 4),(-9,1 % 4)]}
```

Otra cosa que tenemos que tener en cuenta es que todas estas probabilidades deben sumar `1`. Si todo esto son cosas que pueden ocurrir, no tiene sentido que la suma de sus probabilidades sea distinta de `1`. Una moneda que al lanzarla salga un 75% de las veces cruz y un 50% de las veces caras es algo que solo podría en otro extraño universo.

Ahora, la gran pregunta, ¿es una mónada? Dado que las listas son mónadas, parece que esto puede también puede ser una mónada. Primero vamos a pensar en `return` ¿Cómo funciona con las listas? Toma un valor y crea una lista unitaria que lo contenga ¿Qué sucederá aquí? Bueno, como se supone que es el contexto mínimo por defecto, también creará una lista unitaria ¿Qué hay de la probabilidad? Bueno, `return x` se supone que siempre crea un valor monádico capaz de albergar `x` como resultado, así que no tiene sentido que su probabilidad sea `0`. Como siempre devuelve el mismo resultado, su probabilidad debe ser `1`.

¿Qué pasa con `>>=`? Parece algo complicado, así que vamos a utilizar el hecho de que `m >>= f` siempre sea igual a `join (fmap f m)` para todas las mónadas y centrarnos en como unir una lista de probabilidades que contiene listas de probabilidades. Como ejemplo vamos a considerar una lista donde existe un 25% de probabilidades de que ocurra `'a'` o `'b'`. Tanto `'a'`como `'b'` tienen la misma probabilidad de ocurrir. También existe un 75% de probabilidades de que salga `'c'` o `'d'`. Tanto `'c'` como `'d'`tienen la misma probabilidad de ocurrir. Aquí tienes una imagen que representa este posible escenario.



¿Cuáles son las probabilidades de cada uno de estos valores ocurra? Si dibujamos todas estas probabilidades como cajas, cada una con una probabilidad, ¿cuáles serían estas probabilidades? Para calcularlas, todo lo que tenemos que hacer es multiplicar la probabilidad con la probabilidad que lo contenga. `'a'` tendrá una probabilidad de un octavo, al igual que`'b'`, ya que si multiplicamos un medio por un cuarto obtenemos un octavo.`'c'` ocurrirá tres octavos de las veces porque si multiplicamos un medio por tres cuartos obtenemos tres octavos. `'d'` ocurrirá tres octavos. Si sumamos todas las probabilidades dará uno como resultado.

Así se representaría la situación actual:

```
thisSituation :: Prob (Prob Char)
thisSituation = Prob
    [( Prob [('a',1%2),('b',1%2)] , 1%4 )
    ,( Prob [('c',1%2),('d',1%2)] , 3%4)
    ]
```

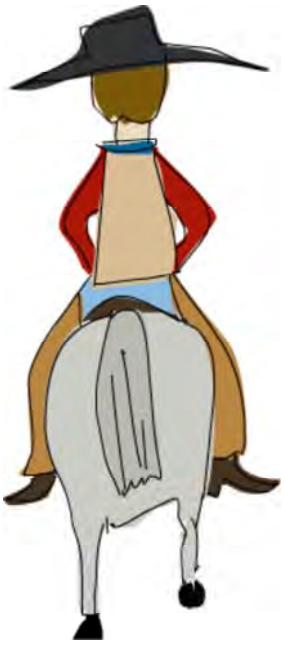
Fíjate que el tipo es `Prob (Pron Char)`. Así que ahora que hemos averiguado como unir varias listas de probabilidades anidadas, solo tenemos que escribir el código y utilizarlo para implementar `>>=` como `join (fmap f m)` y ya tendremos nuestra mónada lista. Aquí tenemos la función `flattern`, tiene este nombre porque alguien ya utilizo antes que nosotros el nombre `join`:

```
flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concat $ map multAll xs
    where multAll (Prob innerxs,p) = map (\(x,r) -> (x,p*r)) innerxs
```

La función `multAll` toma un dupla con una lista de probabilidades y una probabilidad `p` y produce una nueva lista de probabilidades en las que las probabilidades de todos los elementos han sido multiplicadas por `p`. Mapeamos `multAll` sobre cada dupla de la lista anidada y luego la aplanamos con `concat`.

Ahora que ya tenemos todo lo que necesitamos podemos escribir la instancia de`Monad`.

```
instance Monad Prob where
    return x = Prob [(x,1%1)]
    m >>= f = flatten (fmap f m)
    fail _ = Prob []
```



Como ya hemos hecho todo el trabajo sucio, la instancia es muy simple. También hemos definido la función `fail`, que al igual que la instancia de la listas, devuelve una lista vacía. Así que si ocurre un fallo en un ajuste de patrones dentro de una expresión `do`, ocurrirá un fallo en el contexto en si mismo.

Es importante también comprobar si se cumple la leyes de las mónadas. La primera ley dice que `return x >>= f` debe ser igual que `f x`. Una prueba rigurosa sería algo tediosa, pero podemos ver que si tomamos un valor y lo introducimos en contexto mínimo por defecto con `return` y luego mapeamos una función con `fmap` sobre este valor para después aplanar la lista de probabilidades resultante, cada probabilidad que produzca la función será multiplicada por `1%1`, así que el contexto no se verá afectado. El razonamiento por el cual `m >>= return` es igual a `m` es similar. La tercera ley dice que `f <=< (g <=< h)` debe ser igual a `(f <=< g) <=< h`. Esta ley también se cumple ya que mantenemos la mónada lista como base de esta nueva mónada y por que la multiplicación es asociativa. `1%2 * (1%3 * 1%5)`es igual que `(1%2 * 1%3) * 1%5`.

Ahora que tenemos una mónada, ¿qué podemos hacer con ella? Bueno, nos puede ayudar a hacer cálculos con probabilidades. Podemos tratar eventos probabilísticos como valores con un cierto contexto y la mónada probabilidad se encargará de que las probabilidades se reflejen en el resultado final.

Digamos que tenemos dos monedas normales y una moneda trucada que siempre saca cruz nueve de cada diez veces. Si lanzamos todas estas monedas a la vez, ¿cuáles son las probabilidad de que todas ellas sean cruz? Primero, vamos a crear unas listas de probabilidades para las monedas normales y para la trucada:

```
data Coin = Heads | Tails deriving (Show, Eq)

coin, loadedCoin :: Prob Coin
coin = Prob [(Heads,1%2),(Tails,1%2)]

loadedCoin = Prob [(Heads,1%10),(Tails,9%10)]
```

Luego creamos la acción de lanzar las monedas:

```
import Data.List (all)

flipThree :: Prob Bool
flipThree = do
    a <- coin
    b <- coin
```

```
    c <- loadedCoin
    return (all (==Tails) [a,b,c])
```

Vemos que las probabilidades de que todas ellas sean cara no son muy buenas, incluso aunque tengamos una moneda trucada.

```
ghci> getProb flipThree
[(False,1 % 40),(False,9 % 40),(False,1 % 40),(False,9 % 40),
 (False,1 % 40),(False,9 % 40),(False,1 % 40),(True,9 % 40)]
```

Las tres serán cruz nueve veces de cuarenta lanzamientos, lo cual es menos del 25%. Podemos ver que la mónada no sabe como unir todos los valores iguales a `False`, donde no todas las tres monedas fueron cruz. No es un gran problema, ya que podemos crear una función que tome elemento a elemento y vaya sumando las probabilidades del mismo suceso. Ya tienes algo que hacer.

En esta sección hemos pasado de tener una pregunta (¿qué pasaría si añadimos información sobre la probabilidad de un elemento?) a crear un tipo, identificando una mónada y finalmente creando una instancia para trabajar con ella. Creo que hemos hecho bastante. A esta alturas ya deberíamos tener una buena idea que son las mónadas y como trabajar con ellas.

Zippers

Mientras que la pureza de Haskell nos da un montón de beneficios, nos hace abordar algunos problemas de forma muy diferente a como lo haríamos en otros lenguajes impuros. Debido a la transparencia referencial de Haskell, un valor es exactamente igual a otro si ambos representan lo mismo.

Si tenemos tres árboles llenos de cincos y queremos cambiar uno de ellos a seis, tenemos que tener algún modo de decir qué cinco en concreto del árbol queremos modificar. Tenemos que conocer la posición que ocupa en el árbol. En los lenguajes imperativos podemos ver en que parte de la memoria se encuentra el cinco que queremos modificar y ya esta. Pero en Haskell, un cinco es exactamente igual a cualquier otro cinco, así que no podemos elegir uno basándonos en que posición ocupa en la memoria. Tampoco podemos *cambiar*nada. Cuando decimos que vamos a modificar un árbol, en realidad significa que vamos a tomar un árbol y devolver uno nuevo que será similar al original, pero algo diferente.

Una cosa que podemos hacer es recordar el camino que seguimos para llegar al elemento que queremos modificar desde la raíz del árbol. Podríamos decir, toma este árbol, vez a la izquierda, ves a la derecha, vuelve a ir a la izquierda y modifica el elemento que se encuentre allí. Aunque esto funcionaría, puede ser ineficiente. Si luego queremos modificar un elemento que se encuentra al lado del elemento que acabamos de modificar, tenemos que recorrer de nuevo todo el camino empezando por la raíz.

En este capítulo veremos como podemos tomar una estructura de datos cualquiera y centrarnos en la forma en la que modificamos y nos desplazamos por sus elementos de forma eficiente.

Dando un paseo

Como aprendimos en clase de ciencias naturales, existen mucho tipos de árboles diferentes, así que vamos a elegir una semilla y plantar el nuestro. Aquí la tienes:

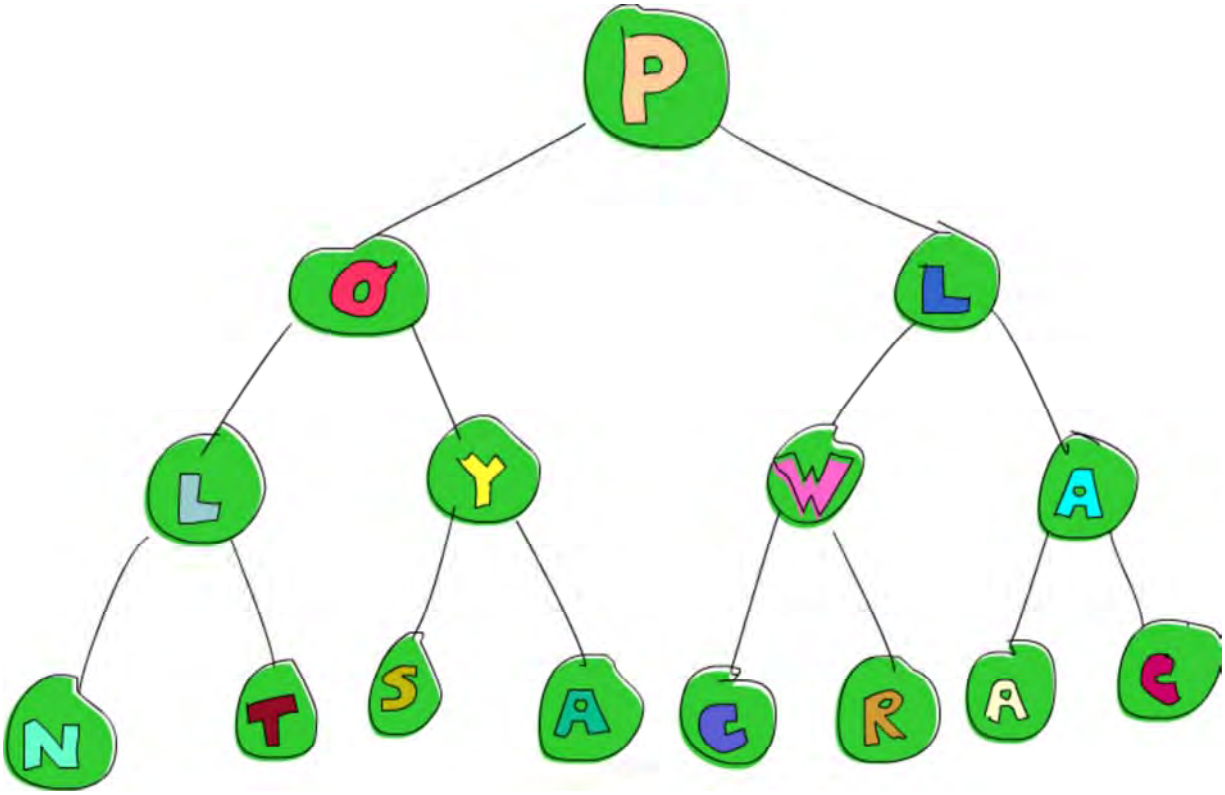
```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Así que este árbol es o bien `Empty` o bien es un nodo que contiene dos sub-árboles. Aquí tienes un ejemplo de árbol de este tipo, ¡gratis!

```
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
```

```
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

Y así es su representación gráfica/artística:



¿Ves esa `W`? Digamos que queremos cambiarla por una `P` ¿Cómo lo hacemos? Bueno, una forma sería utilizando un ajuste de patrones sobre el árbol hasta que encontremos el elemento que buscamos, es decir, vamos por la derecha, luego por la izquierda y modificamos el elemento. Así sería:

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

¡Aarg! No solo es feo si no también algo confuso ¿Qué hace esto? Bueno, utilizamos un ajuste de patrones sobre el árbol y llamamos a su elemento raíz`x` (que en este caso será `'P'`) y su sub-árbol izquierdo `l`. En lugar de dar un nombre a su sub-árbol derecho, utilizamos otro patrón sobre él. Continuamos ese ajuste de patrones hasta que alcanzamos el sub-árbol cuya raíz es `'W'`. Una vez hemos llegado, reconstruimos el árbol, solo que en lugar de que ese sub-árbol contenga una `'W'` contendrá una `'P'`.

¿Existe alguna forma de hacer esto mejor? Podríamos crear una función que tome un árbol junto a una lista de direcciones. Las direcciones será o bien `L` (izquierda) o bien `R` (derecha), de forma que cambiamos el elemento una vez hemos seguido todas las direcciones.

```
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions-> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
```

```
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Si el primer elemento de la lista de direcciones es `L`, creamos un árbol que igual al anterior solo que su sub-árbol izquierdo ahora contendrá el elemento modificado a `P`. Cuando llamamos recursivamente a `changeToP`, le pasamos únicamente la cola de la listas de direcciones, porque sino volvería a tomar la misma dirección. Hacemos lo mismo en caso de `R`. Si la lista de direcciones está vacía, significa que hemos alcanzado nuestro destino, así que devolvemos un árbol idéntico al que hemos recibido, solo que este nuevo árbol tendrá `'P'` como elemento raíz.

Para evitar tener que mostrar el árbol entero, vamos a crear una función que tome una lista de direcciones y nos devuelva el elemento que se encuentra en esa posición.

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

Esta función es muy parecida a `changeToP`, solo que en lugar de reconstruir el árbol paso a paso, ignora cualquier cosa excepto su destino. Vamos a cambiar `'W'` por `'P'` y luego comprobaremos si el árbol se ha modificado correctamente:

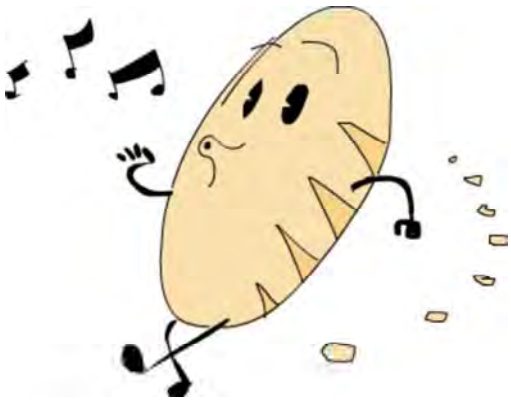
```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

Genial, parece que funciona. En estas funciones, la lista de direcciones actúa como especie de objetivo, ya que señala un sub-árbol concreto del árbol principal. Por ejemplo, una lista de direcciones como `[R]` señala el sub-árbol izquierdo que cuelga de la raíz. Una lista de direcciones vacía señala el mismo árbol principal.

Aunque estas técnicas parecen correctas, pueden ser más bien ineficientes, especialmente si queremos modificar elementos de forma repetida. Digamos que tenemos un árbol inmenso y una larga lista de direcciones que señala un elemento que se encuentra al final del árbol. Utilizamos esta lista de direcciones para recorrer el árbol y modificar dicho elemento. Si queremos modificar un elemento que se encuentra cerca del elemento que acabamos de modificar, tenemos que empezar otra ves desde la raíz del árbol y volver a recorrer de nuevo todo el camino.

En la siguiente sección veremos un forma mejor de señalar un sub-árbol, una que nos permita señalar de forma eficiente a los sub-árbol vecinos.

Un rastro de migas



Vale, así que para centrarnos o señalar un solo sub-árbol, buscamos algo mejor que una simple lista de direcciones que parta siempre desde la raíz ¿Ayudaría si comenzamos desde la raíz y nos movemos a la izquierda o la derecha y al mismo tiempo dejáramos una especie de rastro? Es decir, si vamos a la izquierda, recordamos que hemos ido por la izquierda, y si vamos por la derecha, recordamos que hemos ido por la derecha. Podemos intentarlo.

Para representar este rastro, usaremos también una lista de direcciones (es decir, o bien `L` o bien `R`), solo que en lugar de llamarlo`Directions` (direcciones) lo llamaremos `Breadcrumbs` (rastro), ya que iremos dejando las direcciones que hemos tomado a lo largo del camino.

```
type Breadcrumbs = [Direction]
```

Aquí tienes una función que toma un árbol y un rastro y se desplaza al sub-árbol izquierdo añadiendo `L` a la cabeza de la lista que representa el rastro:

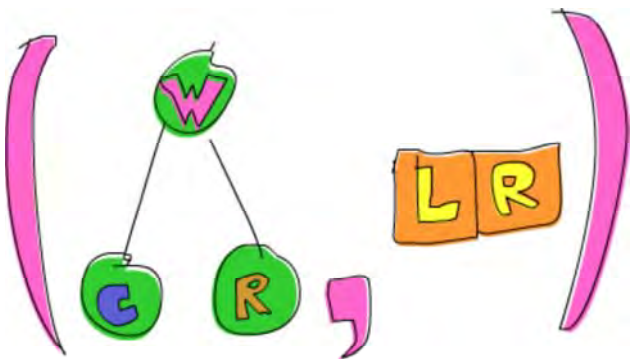
```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

Ignoramos el elemento raíz y el sub-árbol derecho y simplemente devolvemos el sub-árbol izquierdo junto al rastro anterior añadiéndole `L`. Aquí tienes la función que se desplaza a la derecha:

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

Funciona del mismo modo. Vamos a utilizar estas funciones para tomen el árbol `freeTree` y se desplacen primero a la derecha y luego a la izquierda.

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```



Vale, ahora tenemos un árbol que tiene `'W'` como elemento raíz, `'C'`como sub-árbol izquierdo y `'R'` como sub-árbol derecho. El rastro es`[L,R]` porque primero fuimos a la derecha y luego a la izquierda.

Para que recorrer el árbol sea más cómodo vamos crear la función `-:` que definiremos así:

```
x -: f = f x
```

La cual nos permite aplicar funciones a valores escribiendo primero el valor, luego `-:` y al final la función. Así que en lugar de hacer`goRight (freeTree, [])`, podemos escribir `(freeTree, []) -: goRight`. Usando esta función podemos reescribir el código anterior para parezca más que primero vamos a la derecha y luego a la izquierda:

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

Volviendo atrás

¿Qué pasa si queremos volver por el camino que hemos tomado? Gracias al rastro sabemos que el árbol actual es el sub-árbol izquierdo del sub-árbol derecho que colgaba del árbol principal, pero nada más. No nos dice nada acerca del padre del sub-árbol actual para que podamos volver hacia arriba. Parece que aparte del las direcciones que hemos tomado, el rastro también debe contener toda la información que desechamos por el camino. En este caso, el sub-árbol padre que contiene también el sub-árbol izquierdo que no tomamos.

En general, un solo rastro debe contener toda la información suficiente para poder reconstruir el nodo padre. De esta forma, tenemos información sobre todas las posibles rutas que no hemos tomado y también conocemos el camino que hemos tomado, pero debe contener información acerca del sub-árbol en el que nos encontramos actualmente, si no, estaríamos duplicando información.

Vamos a modificar el tipo rastro para que también contenga la información necesaria para almacenar todos los posibles caminos que vamos ignorando mientras recorremos el árbol. En lugar de utilizar `Direction`, creamos un nuevo tipo de datos:

```
data Crumb a = LeftCrumb a (Tree a) | RightCrumb a (Tree a) deriving (Show)
```

Ahora, en lugar de tener únicamente `L`, tenemos `LeftCrumb` que contiene también el nodo desde el cual nos hemos desplazado y el sub-árbol derecho que no hemos visitado. En lugar de `R`, ahora tenemos `RightCrumb` que contiene el nodo desde el cual nos hemos desplazado y el sub-árbol izquierdo que hemos ignorado.

Ahora estos rastros contienen toda la información necesaria para reconstruir el árbol que estamos recorriendo. Así que en lugar de ser un rastro normal, son como una especie de discos de datos que vamos dejando por el camino, ya que contienen mucha más información a parte del camino tomado.

Básicamente, ahora cada rastro es como un sub-árbol cojo. Cuando nos adentramos en un árbol, el rastro almacena toda la información del nodo que nos alejamos exceptuando el sub-árbol que estamos recorriendo. También tenemos que fijarnos en la información que vamos ignorando, por ejemplo, en caso de `LeftCrumb` sabemos que nos acabamos de desplazar por el sub-árbol izquierdo, así que no guardamos ninguna información de este sub-árbol.

Vamos a modificar el sinónimo de tipo `Breadcrumbs` para refleje este cambio:

```
type Breadcrumbs a = [Crumb a]
```

A continuación vamos modificar las funciones `goLeft` y `goRight` para que almacenen en el rastro la información de los caminos que no hemos tomado, en lugar de ignorar esta información como hacíamos antes. Así sería`goLeft`:

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Es muy parecida a la versión anterior de `goLeft`, solo que en lugar de añadir `L` a la cabeza de la lista de rastros, añadimos un elemento`LeftCrumb` para representar que hemos tomado el camino izquierdo y además indicamos el nodo desde el que nos hemos desplazado (es decir `x`) y el camino que no hemos tomado (es decir, el sub-árbol derecho, `r`).

Fíjate que esta función asume que el árbol en el que nos encontramos no es`Empty`. Un árbol vacío no tiene ningún sub-árbol, así que si intentamos movernos por un árbol vacío, obtendremos un error a la hora de ajustar los patrones.

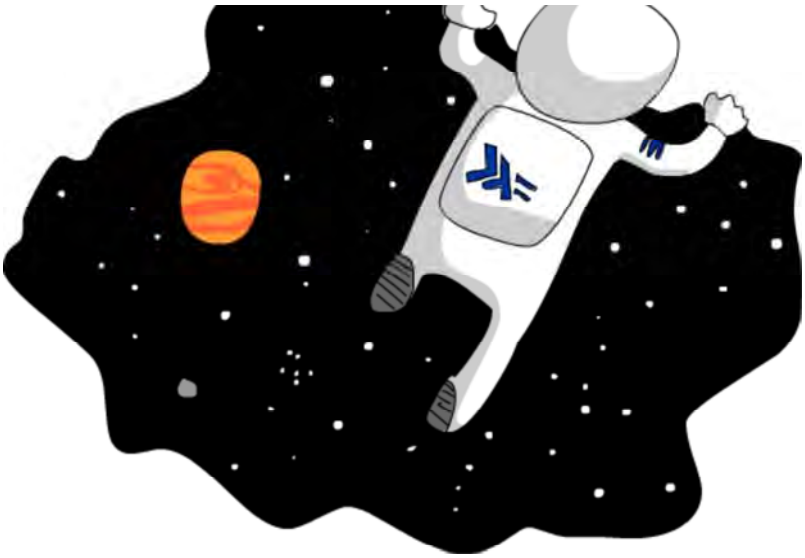
`goRight` es parecido:

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

Ahora somos totalmente capaces de movernos de izquierda a derecha. Lo que aún no podemos hacer es volver por el camino recorrido utilizando la información que indica los nodos padres que hemos recorrido. Aquí tienes la función`goUp`:

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

No encontramos en el árbol `t` y tenemos que comprobar el último `Crumb`. Si es un `LeftCrumb`, entonces



reconstruimos un nuevo árbol donde `t` es el sub-árbol izquierdo y utilizamos la información del sub-árbol derecho que no hemos visitado junto al elemento del nodo padre para reconstruir un nuevo`Node`. Como hemos utilizado el rastro anterior para recrear el nuevo nodo, por decirlo de algún modo, la lista de rastros ya no tiene que contener este último rastro.

Fíjate que esta función genera un error en caso que ya nos encontremos en la cima del árbol. Luego veremos como utilizar la mónada `Maybe` para representar los posibles fallos de desplazamiento.

Gracias al par formado por `Tree a` y `Breadcrumbs a`, tenemos toda la información necesaria para reconstruir el árbol entero y también tenemos señalado un nodo concreto. Este modelo nos permite también movernos fácilmente hacia arriba, izquierda o derecha. Todo par que contenga una parte seleccionada de una estructura y todo la parte que rodea a esa parte seleccionada se llama *zipper*, esto es así porque se parece a la acción de aplicar `zip` sobre listas normales de duplas. Un buen sinónimo de tipo sería:

```
type Zipper a = (Tree a, Breadcrumbs a)
```

Preferiría llamar al sinónimo de tipos `Focus` ya que de esta forma es más claro que estamos seleccionando una parte de la estructura, pero el termino*zipper* se utiliza ampliamente, así que nos quedamos con `Zipper`.

Manipulando árboles seleccionados

Ahora que nos podemos mover de arriba a abajo, vamos a crear una función que modifique el elemento raíz del sub-árbol que seleccione un *zipper*.

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Si estamos seleccionando un nodo, modificamos su elemento raíz con la función`f`. Si estamos seleccionando un árbol vacío, dejamos éste como estaba. Ahora podemos empezar con un árbol, movernos a donde queramos y modificar un elemento, todo esto mientras mantenemos seleccionado un elemento de forma que nos podemos desplazar fácilmente de arriba a abajo. Un ejemplo:

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

Vamos a la izquierda, luego a la derecha y luego remplazamos el elemento raíz del sub-árbol en el que nos encontramos por `'P'`. Se lee mejor si utilizamos `-:`:

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

Luego podemos desplazarnos hacía arriba y remplazar el elemento por una misteriosa `'X'`:

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

O con `-:`:

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Movernos hacia arriba es fácil gracias a que el rastro que vamos dejando que contiene los caminos que no hemos tomado, así que, es como deshacer el camino. Por esta razón, cuando queremos movernos hacia arriba no tenemos que volver a empezar desde la raíz inicial, simplemente podemos volver por el camino que hemos tomado.

Cada nodo posee dos sub-árboles, incluso aunque los dos sub-árboles sean árboles vacíos. Así que si estamos seleccionando un sub-árbol vacío, una cosa que podemos hacer es remplazar un sub-árbol vació por un árbol que contenga un nodo.

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

Tomamos un árbol y un *zipper* y devolvemos un nuevo *zipper* que tendrá seleccionado el árbol que pasemos como parámetro. Esta función no solo nos permite añadir nodos a las hojas de un árbol, sino que también podemos remplazar sub-árboles enteros. Vamos a añadir un árbol a la parte inferior izquierda de `freeTree`:

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

`newFocus` ahora selecciona un nuevo árbol que ha sido añadido al árbol original. Si utilizáramos `goUp` para subir por el árbol, veríamos que sería igual que `freeTree` pero con un nodo adicional `'Z'` en su parte inferior izquierda.

Me voy a la cima del árbol, donde el aire está limpio y fresco

Crear una función que seleccione la cima del árbol, independientemente del nodo seleccionado, es realmente fácil:

```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

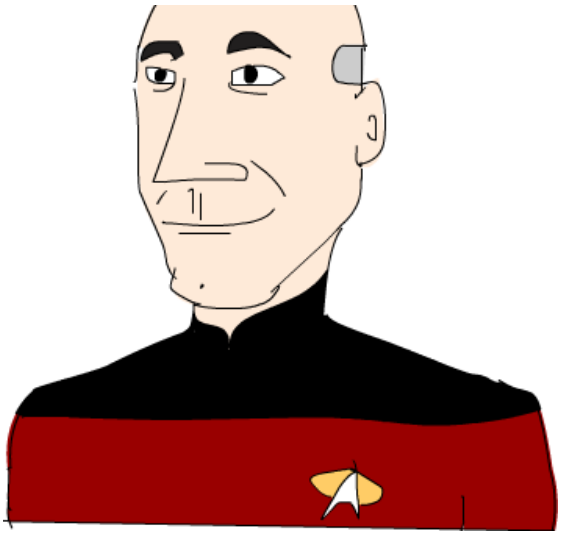
Si nuestro rastro está vacío, significa que ya estamos en la cima del árbol, así que solo tenemos que devolver el mismo nodo que está seleccionado. De otro modo, solo tenemos que seleccionar el nodo padre del actual y volver a aplicar de forma recursiva `topMost`. Ahora podemos dar vueltas por un árbol, ir a la izquierda o a la derecha, aplicar `modify` o `attach` para realizar unas cuantas modificaciones, y luego, gracias a `topMost`, volver a selecciona la raíz principal del árbol y ver si hemos modificado correctamente el árbol.

Seleccionando elementos de la listas

Los *zippers* se pueden utilizar con casi cualquier tipo de estructura, así que no debería sorprendente que también se puedan utilizar con las listas. Después de todo, las listas son muy parecidas a los árboles. El los árboles un nodo puede tener un elemento (o no) y varios sub-árboles, mientras que en las listas un elemento puede tener una sola sub-lista. Cuando implementamos*nuestro propio tipo de listas*, definimos el tipo así:

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

Si lo comparamos con la definición anterior de los árboles binarios podemos observar como las listas pueden definirse como un árbol que solo posee un sun-árbol.



La lista `[1,2,3]` es igual que `1:2:3:[]`. Está formada por la cabeza de la lista, que es `1` y su cola, que es `2:3:[]`. Al mismo tiempo, `2:3:[]` está formado por su cabeza, que es `2`, y por su cola, que es `3:[]`. `3:[]` está formado por su cabeza `3` y su cola que es la lista vacía `[]`.

Vamos a crear un *zipper* para las listas. Para modificar el elemento seleccionado de una lista, podemos mover hacia adelante o hacia atrás (mientras que con los árboles podíamos movernos a la derecha, a la izquierda, y arriba). La parte que seleccionábamos con los árboles era un sub-árbol, a la vez que el rastro que dejábamos cuando avanzábamos. Ahora, ¿qué tendremos que dejar como rastro? Cuando estábamos trabajando con árboles binarios, vimos que el rastro tenía que albergar el elemento raíz de su nodo padre junto a todos los sub-árboles que recorrimos. También teníamos que recordar si habíamos ido por la izquierda o por la derecha. Resumiendo, teníamos que poseer toda la información del nodo que contenía el sub-árbol que estábamos seleccionando.

Las listas son más simples que los árboles, así que no tenemos que recordar si hemos ido por la derecha o por la izquierda, ya que solo podemos avanzar en una dirección. Como solo hay un posible sub-árbol para cada nodo, tampoco tenemos que recordar el camino que tomamos. Parece que lo único que debemos recordar el elemento anterior. Si tenemos una lista como `[3,4,5]` y sabemos que el elemento anterior es `2`, podemos volver atrás simplemente añadiendo dicho elemento a la cabeza de la lista, obteniendo así `[2,3,4,5]`.

Como cada rastro es un elemento, no necesitamos crear un nuevo tipo de datos como hicimos con el tipo de datos `Crumb` para los árboles:

```
type ListZipper a = ([a],[a])
```

La primera lista representa la lista que estamos seleccionando y la segunda lista es la lista de rastros. Vamos a crear las funcionen que avancen y retrocedan por las listas:

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Cuando avanzamos, seleccionamos la cola de la lista actual y dejamos la cabeza como rastro. Cuando retrocedemos, tomamos el último rastro y lo insertamos al principio de la lista.

Aquí tienes un ejemplo de estas funciones en acción:

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs,[])
([2,3,4],[1])
ghci> goForward ([2,3,4],[1])
([3,4],[2,1])
ghci> goForward ([3,4],[2,1])
([4],[3,2,1])
ghci> goBack ([4],[3,2,1])
([3,4],[2,1])
```

Podemos observar que el rastro de una listas no es nada más que la parte invertida de la lista que hemos dejado atrás. El elemento que dejamos atrás siempre pasa a formar parte de la cabeza de los rastros, así que es fácil movernos hacía atrás tomando simplemente el primer elemento de los rastros y añadiéndolo a la lista que tenemos seleccionada.

Si estamos creando un editor de texto, podemos utilizar una lista de cadenas para representar las líneas de texto del fichero que estemos editando, luego podemos utilizar un *zipper* de forma que sepamos donde se encuentra el cursor. El hecho de utilizar los *zipper* también facilitaría la introducción de líneas de texto nuevas en cualquier parte del texto o barrar líneas existentes.

Un sistema de ficheros simple

Ahora que sabemos como funcionan los *zippers*, vamos utilizar un árbol para representar un sistema de ficheros y luego crearemos un *zipper* para ese sistema, lo cual nos permitirá movernos entre los directorios de la misma forma que hacemos nosotros mismos.

Si tomamos una versión simplificada de los sistemas de ficheros jerárquicos, podemos observar que básicamente están formados por ficheros y directorios. Los ficheros son las unidades de información y poseen un nombre, mientras que los directorios se utilizan para organizar estos ficheros y pueden contener tanto ficheros como otros directorios. Así que vamos a decir que un objeto de sistema de ficheros es o bien un fichero, que viene acompañado de un nombre y unos datos, o bien un directorio, que viene acompañado de un nombre y un conjunto de objetos que pueden ser tanto ficheros como directorios. Aquí tienes el tipo de datos para este sistema junto un par de sinónimos de tipo:

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

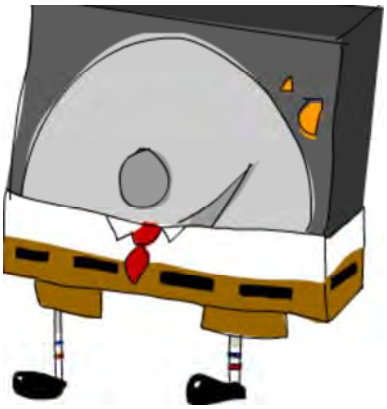
Cada fichero viene con dos cadenas, una representa su nombre y otra sus contenidos. Cada directorio viene con una cadena que representa su nombre y un lista de objetos. Si la lista está vacía, entonces tenemos un directorio vacío.

Aquí tienes un ejemplo:

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl_bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
        ]
```

En verdad es el contenido de mi disco duro en este momento.

Un zipper para el sistema de ficheros



Ahora que tenemos un sistema de ficheros, lo que necesitamos es un *zipper*de forma que podamos desplazarnos, crear, modificar o eliminar ficheros al vez que directorios. De la misma forma que con los árboles binarios y las listas, vamos a ir dejando un rastro que contenga todas las cosas que no hemos visitado. Como ya hemos dicho, cada rastro debe ser una especie de nodo, solo que no debe contener el sub-árbol que estamos seleccionando para no repetir información. También tenemos que tener en cuenta la posición en la que nos encontramos, de forma que podamos volver atrás.

En este caso en particular, el rastro será algo parecido a un directorio, solo que no debe contener el directorio en el que estamos ¿Y porqué no un fichero? Te estarás preguntando. Bueno, porque una vez hemos seleccionado un fichero, no podemos avanzar en el sistema de ficheros, así que no tiene mucho sentido dejar algo en el rastro que diga que venimos de un fichero. Un fichero es algo parecido a un árbol vacío.

Si nos encontramos en el directorio `"root"` y queremos seleccionar el fichero `"dijon_poupon.doc"`, ¿qué debería contener el rastro? Bueno, debería contener el nombre del directorio padre junto con todos los elementos anteriores al fichero que estamos seleccionando más los elementos posteriores. Así que lo que necesitamos es un `Name` y dos listas de objetos. Manteniendo dos listas separadas de elementos, una con los elementos anteriores y otra con los elementos posteriores, sabremos exactamente que seleccionar si volvemos atrás.

Aquí tenemos el tipo rastro para nuestro sistema de ficheros:

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

Y aquí nuestro sinónimo de tipo para *zipper*:

```
type FSZipper = (FSItem, [FSCrumb])
```

Volver atrás por esta jerarquía es muy fácil. Solo tenemos que tomar el último elemento del rastro y seleccionar un nuevo elemento a partir del objeto actualmente seleccionado y del rastro. Así:

```
fsUp :: FSZipper -> FSZipper
fsUp (item, (FSCrumb name ls rs):bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

Como el rastro contiene el nombre del directorio padre, así como los elementos anteriores al objeto seleccionado (es decir, `ls`) y los posteriores (`rs`), retroceder es muy sencillo.

¿Y si queremos avanzar por el sistema de ficheros? Si estamos en `"root"` y queremos seleccionar `"dijon_poupon.doc"`, el rastro contendrá el nombre`"root"` junto con los elementos que preceden a `"dijon_poupon.doc"` y los que van después.

Aquí tienes una función que, dado un nombre, selecciona el fichero o directorio que este contenido en el directorio actual:

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

`fsTo` toma un `Name` y un `FSZipper` y devuelve un nuevo `FSZipper`que tendrá seleccionado el fichero con el nombre dado. El dicho debe estar en el directorio actual. Esta función no busca el fichero sobre todos los directorios, solo con el directorio actual.

Primero utilizamos `break` par dividir la lista de elementos en un lista con los elementos anteriores al fichero que estamos buscando y en una lista con los que van después. Si recuerdas, `break` toma un predicado y una lista y devuelve una dupla que contiene dos listas. La primera lista en la dupla contiene los elementos en los que el predicado no se cumplió. Luego, una vez encuentra un elemento que cumple el predicado, introduce ese elemento y el resto de la lista en la segunda componente de la dupla. Hemos creado un función auxiliar llamada `nameIs` que toma un nombre y un objeto del sistema de ficheros y devuelve `True` si coinciden los nombres.

Ahora, `ls` es una lista que contiene los elementos que preceden al objetos que estamos buscando, `item` es dicho objeto y `rs` es la lista de objetos que viene después del objeto en cuestión. Con todo esto, solo tenemos que devolver el objeto que obtuvimos de `break` y crear un rastro con toda la información requerida.

Fíjate que si el nombre que estamos buscando no está en el directorio actual, el patrón `item:rs` no se ajustará y por lo tanto obtendremos un error. También, si el elemento seleccionado no es directorio, es decir, es un fichero, también obtendremos un error y el programa terminará.

Ahora ya podemos movernos por el sistema de ficheros. Vamos a partir de la raíz y recorrer el sistema hasta el fichero `"skull_man(scary).bmp"`:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

`newFocus` es ahora un *zipper* que selecciona el fichero`"skull_man(scary).bmp"`. Vamos a obtener el primer componente del *zipper*(el objeto seleccionado) y comprobar si es verdad:

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

Vamos a volver atrás y seleccionar su fichero vecino “watermelon_smash.gif”:

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

Manipulando el sistema de ficheros

Ahora que ya podemos navegar por el sistema de ficheros, manipular los elementos es muy fácil. Aquí tienes un función que renombra el fichero o directorio actual:

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

Podemos renombrar el directorio `"pics"` a `"cspi"`:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

Nos hemos metido en el directorio `"pics"`, lo hemos renombrado, y luego hemos vuelto.

¿Qué tal una función que crea un nuevo elemento en el directorio actual?

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

Facilísimo. Ten en cuenta que esta función fallara si intentamos añadir un elemento a algo que no sea un directorio.

Vamos a añadir un fichero a nuestro directorio `"pics"` y luego volver atrás:

```
ghci> let newFocus = (myDisk,[]) -: fsTo "pics" -: fsNewFile (File "heh.jpg" "lol") -: fs
```

Lo realmente interesante de este método es que cuando modificamos el sistema de ficheros, en realidad no modifica ese mismo sistema, si no que devuelve uno totalmente nuevo. De este modo, podremos acceder al sistema de ficheros antiguo (`myDisk` en este caso) y también al nuevo (el primer componente de`newFocus`). Así que gracias a los *zippers*, obtenemos automáticamente copias de diferentes versiones, de forma que siempre podremos referenciar a versiones antiguas aunque lo hayamos modificado. Esto no es una propiedad única de los *zippers*, si no de todas las estructuras de datos de Haskell ya que son inmutables. Sin embargo con los *zippers*, ganamos la habilidad de recorrer y almacenar eficientemente estas estructuras de datos.

Vigila tus pasos

Hasta ahora, cuando recorríamos estructuras de datos, ya sean árboles binarios, listas o sistemas de ficheros, no nos preocupábamos de sí tomábamos un paso en falso y nos salíamos de la estructura. Por ejemplo, la función`goLeft` toma un *zipper* de un árbol binario y mueve el selector al árbol izquierdo:

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Pero, ¿y si el árbol en el que nos encontramos está vacío? Es decir, no es un`Node` si no un `Empty`. En este caso, obtendremos un error de ejecución ya que el ajuste de patrones fallará ya que no hay ningún patrón que se ajuste a árboles vacíos, lo cuales no contienen ningún sub-árbol. Hasta ahora, simplemente hemos asumido que nunca íbamos a intentar seleccionar el sub-árbol izquierdo de un árbol vacío ya que dicho sub-árbol no existe. De todos modos, ir al sub-árbol izquierdo de un árbol vacío no tiene mucho sentido, y hasta ahora no nos hemos preocupado de ello.

O, ¿qué pasaría si estamos en la raíz de un árbol y no tenemos ningún rastro e intentamos continuar hacía arriba? Ocurriría lo mismo. Parece que cuando utilizamos los *zipper*, cada paso que demos puede ser el último (reproducir música siniestra aquí). En otras palabras, cada movimiento puede ser un éxito, pero también fallo. Sí, es la ultima vez que te lo pregunto, y se que lo estás deseando, ¿a qué te recuerda esto? Por supuesto, ¡mónadas! en concreto la mónada `Maybe` que se encarga de contextos con posibles fallos.

Vamos a utilizar la mónada `Maybe` para añadir el contexto de un posible fallo a nuestro pasos. Vamos a tomar las funciones que ya funcionan con el *zipper* de árboles binarios y vamos a convertirlas en funciones monádicas. Primero, vamos a añadir el

contexto de un posible fallo a `goLeft` y`goRight`. Hasta ahora, el fallo de una función se reflejaba en su resultado y no va ser distinto aquí.

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

¡Genial! Ahora si intentamos dar un paso a la izquierda por un árbol vacío obtendremos un `Nothing`.

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

Parece que funciona ¿Y si vamos hacia arriba? Aquí el problema está en si queremos ir hacía arriba y no hay ningún rastro más, ya que esta situación indica que nos encontramos en la cima del árbol. Esta es la función `goUp`que lanza un error si nos salimos de los límites:

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Y esta la versión modificada:

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

Si tenemos un rastro no hay ningún problema y podemos devolver un nuevo nodo seleccionado. Si embargo, si no hay ningún rastro devolvemos un fallo.

Antes estas funciones tomaban *zippers* y devolvían *zippers*, por lo tanto podíamos encadenarlas así:

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

Ahora, en lugar de devolver un `Zipper a`, devuelven `Maybe (Zipper a)`, así que no podemos encadenar las funciones de este modo. Tuvimos un problema similar cuando estábamos con nuestro*buen amigo el funambulista*, en el capítulo de las mónadas. Él también tomaba un paso detrás de otro, y cada uno de ellos podía resultar en un fallo porque siempre podían aterrizar un grupo de pájaros en lado y desequilibrar la barra.

Ahora el problema lo tenemos nosotros, que somos los que estamos recorriendo el árbol. Por suerte, aprendimos mucho de Pierre y de lo que hizo: cambiar la aplicación normal de funciones por la monádica, utilizando `>>=`, que toma un valor en un contexto (en nuestro caso, `Maybe (Zipper a)`, que representa el contexto de un posible fallo) y se lo pasa a un función de forma que se mantenga el significado del contexto. Así que al igual que nuestro amigo, solo tenemos que intercambiar `-:` por `>>=`. Mira:

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight
```

```
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

Hemos utilizado `return` para introducir un *zipper* en un valor `Just`y luego hemos utilizado `>>=` para pasar ese valor a la función `goRight`. Primero, creamos un árbol que tiene en su rama izquierda un sub-árbol vacío y en su rama derecha dos sub-árbol vacíos. Cuando intentamos ir por la rama derecha, el movimiento tiene éxito porque la operación tiene sentido. Volver a ir a la derecha también está permitido, acabamos seleccionando un árbol vacío. Pero si damos un paso más por tercera vez no tendrá sentido, porque no podemos visitar la rama derecha o izquierda de un sub-árbol vacío, por la tanto obtenemos `Nothing`.

Ahora ya tenemos equipadas nuestras funciones con una red de seguridad que nos salvará si nos caemos. Momento metafórico.

El sistema de fichero también posee un montón de casos donde podría fallar, como intentar seleccionar un fichero o un directorio que no existe. Como último ejercicio, si quieres claro, puedes intentar añadir a estas funciones el contexto de un posibles fallos utilizando la mónada `Maybe`.