Por qué deberías aprender programación funcional ya mismo (con una breve intro a Haskell)

Andrés Marzal

Borrador del día 6 de mayo de 2015

Índice general

1 Creatividad por restricción

La creatividad surge, en ocasiones, de la imposición de restricciones. En poesía, por ejemplo, algunas composiciones deben obedecer restricciones que afectan al número de sílabas por verso, a la rima, al ritmo... Paradójicamente, la observación estricta de las reglas puede favorecer la creatividad: el proceso de selección de palabras, de construcción de frases, de estructura en la exposición de ideas ha de explorar más opciones y debe ser más selectivo. El sometimiento a las reglas *puede* dar lugar a hallazgos estéticos y semánticos. Y, en cualquier, ejercitarse en la creación sometida a reglas es una forma de mejorar de las habilidades creativas en un contexto sin reglas.

1.1 Oulipo

En los años 60 se creó Oulipo (*Ouvrier de littérature potentielle*), un taller literario que proponía la escritura de textos siguiendo restricciones (*contraintes*). Oulipo nace del encuentro entre un escritor, Raymond Queneau, y un químico y divulgador científico/matemático, François Le Lionnais. El grupo contó con escritores reconocidos, como Georges Perec o Italo Calvino, pero también con matemáticos y pintores. Sus obras, especialmente las que siguen las reglas más peregrinas, son un ejemplo de creatividad por imposición de restricciones.

1.2 Algunos formatos de Oulipo

Algunos de los formatos literarios que propone Oulipo son:

- **Bola de nieve**: cada línea solo puede contener una palabra, y cada palabra ha de tener una letra más que la anterior.
- **Lipograma**: prohibición de usar ciertas letras en el texto[1]. Un tipo de lipograma específico, denominado "del prisionero" o "de Macao", es el que prohíbe usar letras con ascendentes y descendentes tipográficos (b, d, f, g, h, j, k, l, p, q, t, y).
- **Palíndromo**: soneto construido con técnicas palindrómicas.
- **Univocalismo**: poema construido usando consonantes y solo una de las cinco vocales.

Hay un listado bastante completo en http://oulipo.net/fr/contraintes.

1.3 Ou*po

Oulipo dejó huella e influyó en autores ajenos al grupo como Cortázar. Pero, además dio lugar a varias ramificaciones:

- Oulipopo: taller de literatura policiaca potencial.
- Oumupo: taller de música potencial.
- Oupeinpo: taller de pintura (peinture) potencial.
- Oucinépo: taller de cinematografía (cinématographie) potencial.
- Oucuipo: taller de cocina (cuisine) potencial.
- Oubapo: taller de cómic (bande dessinée) potencial.

1.4 Ouprogpo

Imaginemos que nos toca ser pioneros del Ouprogpo (*Ouvrier de programmation potentielle*) y proponemos la construcción de programas con estas restricciones:

- El sistema de tipos es estático.
- No puedes usar bucles ni, en general, estructuras de control de flujo.
- Las funciones tienen un único parámetro.
- Cada función consta de una sola expresión.
- Las funciones no tiene efectos secundarios y, dada una entrada, siempre producen el mismo resultado y nada más que ese resultado.

[1] Estas técnicas no las inventaron los miembros de Oulipo. Jardiel Poncela, por ejemplo, escribió relatos en los años 30 que excluyen el uso de una vocal determinada (por ejemplo, en "Un marido sin vocación..."), no usa la letra más común del castellano: la e). Una de las actividades de Oulipo era, precisamente, buscar "plagiarios adelantados", es decir, gente que plagió a Oulipo antes de que Oulipo existiese.]

- Una vez se asigna un valor a un identificador, no se puede modificar.
- Las operaciones sobre estructuras de datos son no destructivas, es decir, puedo disponer de la versión de la estructura de datos previa a la operación y la versión posterior a la misma.
- El orden de ejecución de las diferentes expresiones no importa.

Este conjunto de restricciones ya tiene nombre: **Programación funcional**.

2 Sales pitch

2.1 Una tendencia

Acaba de aparecer Java 8 (2014), el cambio más importante en el lenguaje desde Java 5 (2004), es decir, en una década. El tema de Java 5 fue la introducción de los tipos genéricos en el lenguaje. Los tipos genéricos provienen de la programación funcional. De hecho, Philip Wadler (co-autor de Haskell) y Martin Odersky (autor de Scala) son dos de los autores de la especificación de los tipos genéricos en Java 5 (además de autores de Generic Java, en 1998, que fue el trabajo seminal para la especificación de *generics* en Java 5.

Java 8 es un cambio "temático". El tema principal es la introducción de aspectos de la programación funcional en el lenguaje:

- lambda-expresiones y clausuras
- funciones de orden más alto (*higher-order functions*)
- streams y soporte para cálculo con modelo map-filter-reduce
- evaluación perezosa
- default methods en interfaces
- tratamiento de errores con mónada (*optional*)

¿Por qué un lenguaje tan conservador hace sus dos movimientos más fuertes en esa dirección? Los cambios recientes en la tecnología aumentan la percepción de que la programación funcional tiene ventajas en términos de productividad para los desarrolladores y/o para la eficiencia de los programas. Por ejemplo, la aparición de sistemas multiprocesador de bajo coste y el freno al aumento velocidad pura de los núcleos individuales hace que la concurrencia resulte más y más atractiva. Pues resulta que la programación impone un conjunto de restricciones que simplifica el diseño de programas concurrentes.

El auge de Scala, un lenguaje JVM interoperable con Java que integra programación funcional y orientación a objetos, evidencia ese aumento de productividad en el desarrollador sin sacrificio de eficiencia en la ejecución. No solo Scala es más expresivo que Java y se presta más fácilmente al diseño de DSL embebidos que Java: además, algunos *frameworks* (Akka, Play...) y librerías (colecciones inmutables, scalaz, Scala Combinator Parsing...) dan acceso a ciertas tecnologías (programación reactiva, concurrencia, actores, parsing aplicativo y monádico...) con mayor facilidad de uso.

Si Java no evoluciona y se posiciona haciendo accesible desde el propio lenguaje esa tecnología, podría perder su posición de dominio en el ecosistema JVM (al menos en lo que toca a los "cool guys").

Este giro reciente hacia lo funcional no es exclusivo de Java 8. Otros lenguajes han integrado aspectos de la programación funcional, bien desde el diseño inicial, bien en revisiones importantes del lenguaje.

- **Python**: listas comprensivas, lambda funciones, librería estándar para aplicar técnicas funcionales, generación perezosa de secuencias.
- **C#**: lambda-funciones, immutability (structs), LINQ, type inference, immutable collections, inicialización perezosa.
- **JavaScript**: lambda-functions, first-class functions, higher-order-functions, map-filter. ECMAScript 6 trae tail-call optimization, pattern matching (destructuring), list (array) comprehension. Hay, además, una librería de apoyo a la programación funcional: Underscore.js.
- **C++**: Inferencia de tipos, lambda funciones...
- **F#**: Lenguaje funcional para .NET
- **Clojure**: LISP para JVM
- **Scala**: Lenguaje funcional + OO para JVM, con lambda-funciones, currying, pattern-matching, genéricos, types y kinds, tipos de datos algebraicos, implicits, colecciones persistentes.
- **Swift**: Copia (parcial) de Scala para el ecosistema Apple
- **Rust**: Rasgos (traits), concordancia de patrones (pattern matching), inferencia de tipos (type inference)
- **Ruby**: bloques como funciones de orden superior, map-filter-reduce, generación perezosa de secuencias.

Otro indicador de la importancia creciente de la programación funcional es la cantidad de publicaciones que se dedican a hablarnos de ella o de cómo implementarla/adoptarla sobre lenguajes de uso corriente:

- Functional thinking, de Neal Ford.
- Functional reactive programming, de Stephen Blackheath y Anthony Jones.
- Functional programming in Scala, de Paul Chiusano y Rúnar Bjarnason.
- Java 8 lambdas: pragmatic functional programming, de Richard Warburton.
- Functional Python programming, de Stelevn Lott.
- Functional programming in PHP, de Simon Holywell.
- Functional programming in Java: harnessing the power of lambda 8 expressions, de Venkat Subramaniam.
- Becoming functional: steps for transforming into a functional programmer, de Joshua Backfield.
- Functional Javascript: introducing functional programming with underscore.js, de Michael Fogus.
- Functional programming in C#, de Oliver Sturm.
- Functional programming in Swift, de Chris Eidhof y Florian Kugler.
- Functional programming in Javascript, de Dan Mantyla.
- Functional programming for the object oriented programmer, de Brian Marick.

Me he limitado a citar unos cuantos libros dedicados a introducir la programación funcional sobre lenguajes puramente funcionales. Hay, también abundante literatura reciente sobre lenguajes que ya nacen con un diseño orientado a la programación funcional, como Haskell, Clojure, Ocaml, F#...

Así pues, lo que empezó como un "desvío académico" que podría haber continuado en ese mundo ha acabado por integrarse en la tecnología de uso común.

Así pues, conviene aprender esa tecnología y aprenderla bien. No basta con usar una o dos técnicas por instinto o por mímesis. Debes aprender los fundamentos y debes ponerlos en práctica. ¿Cómo hacerlo? ¿Puedo aprender esos fundamentos sin renunciar a las herramientas que uso convencionalmente?

Probablemente estés muy cómodo con lenguajes dinámicos (Python, Ruby...), especialmente por su expresividad y comodidad para el prototipado. No te aconsejo su uso para aprender programación funcional. Algunas de las construcciones más interesantes necesitan apoyo del sistema de tipos. Mejor dicho: no tienen sentido o resultan muy rebuscadas si el sistema de tipo "se aparta" en aras del prototipado rápido.

O puede que estés cómodo en el ecosistema JVM. Aunque Java va incorporando elementos de la programación funcional, lo cierto es que sigue resulta verboso. Por otra parte, els sistema de tipos tiene limitaciones que dificultan la expresión de ciertos elementos. (Y las limitaciones de sus *generics* (por culpa del *type erasure*) siempre me han echado atrás.)

En el entorno JVM se te presentarán dos opciones que están teniendo éxito y que, al ser interoperables con Java, minimizan el coste personal de la transición:

- Clojure: un LISP/Scheme para JVM.
- Scala: un mix de orientación a objetos y programación funcional.

Clojure tiene un problema similar al de Python y Ruby: es un lenguaje con tipado dinámico. Si bien aprenderás mucho de programación funcional con Clojure, te quedarás a las puertas de aprender bien un material jugoso, el que hará que tu cerebro "te duela".

Yo aprendí Scala por curiosidad: corre sobre JVM, es interoperable con Java, tan eficiente como Java y con la apariencia de ser un Python++: muy expresivo y con un sistema de tipos que *parece* dinámico (gracias a la inferencia de tipos). En primera instancia, la mezcla de orientación a objetos y programación funcional facilita el aprendizaje.

Al trabajar con Scala y leer blogs o ver conferencias, uno va adentrándose progresivamente en el mundo de la programación funcional. Un día acabas por descubrir la librería Scalaz y no entiendes absolutamente nada, pero percibes que "ahí hay algo". Empezar con Scalaz supone, en algún momento, pararse a curiosear Haskell. Y ahí empieza todo. Como dice Alejandro Stucky y cita Martin Odersky, el autor de Scala, "**Scala is a gateway drug to Haskell**".

La pendiente deslizante que supone iniciarse en el aprendizaje de Haskell te lleva a revisar y aprender conceptos matemáticos que aparecen como patrones en el diseño de programas funcionales. Son patrones por que abstraen la esencia de diseños de solución a problemas en apariencia muy distintos. Dicho así, te parecerá que serán el correlato funcional de los patrones de diseño que conoces de la orientación a objetos. Pero no, resulta que son bestias muy extrañas y diferentes. La percepción que tienes de la programación acaba viéndose afectada por ese bestiario funcional.

> "I've noticed several times when someone says 'X really changed the way I think about programming,' frequently X=Haskell."
>
> Andrew Binstock, editor-in-chief, Dr. Dobb's

2.2 Una venta en diferido

No sé si tiene mucho sentido vender en abstracto las ventajas de la programación funcional. Os puedo decir que tiene muchas ventajas, claro, pero quedaré como un vendedor con su discurso aprendido para colocar un producto. O pareceré un fanboy de la última moda en lenguajes de programación. Hay gente que ha hecho mucha mejor venta de la programación funcional en términos generales de la que yo pueda hacer. Os recomiendo un par de presentaciones y zanjamos el asunto de la venta en abstracto:

- 2 years of real world FP at REA, de Ken Scambler.
- How to sell excellence, de Michael O. Church.

2.3 Fundamentos sólidos

¿Qué hay en la programación funcional que resulta tan atractivo? Lo primero que hay que decir es que está apoyada en dos pilares matemáticos potentes:

- el lambda-cálculo
- la teoría de categorías

2.3.1 Lambda-cálculo

Es un sistema formal que describe el cálculo en términos una abstracción de las funciones y su aplicación. Lo fomuló Alonzo Church en los años 30 del siglo XX y es un modelo de computación universal, como lo es la máquina de Turing.

En las mates aprendidas en el cole denotamos las funciones de un modo que se presta a cierta ambigüedad. Una función que eleva un número a su cuadrado se denota con $f(x) = x^2$. La aplicación de la función se denota encerrando el argumento entre paréntesis. Así, $f(2)$ es la aplicación de la función al valor 2. La ambigüedad surge al usar la expresión $f(x)$. ¿Nos referimos a la función f o a su aplicación a una variable x que puede tener un valor determinado? El lambda-cálculo evita esta y otras ambigüedades. Va más allá de la notación: es un **sistema formal**.

En lambda-cálculo, la función que calcula el cuadrado de un número se denota así:

$$\lambda x.x^2$$

La aplicación de la función se denota poniendo el valor a continuación de la función

$$(\lambda x.x^2)\, 2 = 4$$

Nótese que no hacen falta paréntesis que encierren el argumento y que no hay confusión posible en el caso de una expresión como

$$(\lambda x.x^2)\, x$$

La x que aparece junto a λ denota el parámetro, la x en el cuerpo de la función, también. La última x ha de ser una variable que tiene valor y es ese valor el que suministramos como argumento.

Solo hay tres construcciones en lambda-cálculo:

- Nombre: proporciona un valor (ejemplo: x)
- Definición de una función con un parámetro (ejemplo: $\lambda x.2x$)
- Aplicación de una función a un valor (ejemplo: $(\lambda x.2x)\, 10$)

En realidad, y por concisión, usamos más construcciones, pero que en última instancia se pueden reducir a estas construcciones básicas. Es decir, hay azúcar sintáctico para facilitar la legibilidad, pero todo se puede reducir en última instancia a las construcciones básicas.

Una función con dos parámetros que calcula la suma de sus cuadrados se puede denotar así $\lambda(x, y).x^2 + y^2$. Pero hay una forma más elegante: $\lambda x.\lambda y.x^2 + y^2$. Si usamos un paréntesis en la expresión anterior tenemos la forma $\lambda x.(\lambda y.x^2 + y^2)$, que deja claro que una función con dos parámetros no es más que una función con un parámetro que devuelve una función con un parámetro: $\lambda x.(\lambda y.x^2 + y^2)$. Si aplico la función a un solo argumento, obtengo una función $(\lambda x.(\lambda y.x^2 + y^2))\, 4 = \lambda y.16 + y^2$. Esta segunda función puedo aplicarla a un

nuevo argumento: $(\lambda y.16 + y^2)\, 3 = 25$. No hace falta ir paso a paso: podemos denotar la función a los dos argumentos directamente con $(\lambda x.\lambda y.x^2 + y^2)\, 4\, 3 = 25$.

Decimos de las funciones de n parámetros en ese "formato" de función con un solo parámetro que devuelve una función con $n - 1$ parámetros en ese mismo "formato" está en forma "currificada" (*currified*), por Haskell Curry (de quien toma nombre el lenguaje Haskell).

La currificación se puede expresar con relativa comodidad en lenguajes como Javascript. Estas dos funciones JavaScript, por ejemplo, calculan la suma de dos números:

```
suma = function (x, y) { return x + y; }

sumaCurry = function (x) {
              return function (y) {
                return x + y;
              }
            }
```

Fíjate en la forma diferente de invocar a una y a otra:

```
suma(2, 4)
suma(2)(4)
```

No conviene seguir abundando en el λ-cálculo aquí, pero cuando conozcamos lo básico de Haskell apreciaremos la semejanza. Haskell es, probablemente, el lenguaje que implementa con mayor fidelidad el lambda-cálculo en una variante denominada "lambda-cálculo tipado".

2.3.2 Teoría de categorías

La teoría de categorías es una rama de las matemáticas que estudia estructuras que surgen en colecciones de objetos y flechas entre ellos (o morfismos). Es una especie de herramienta de abstracción capaz de dar resultados en teoría de conjuntos, de anillos, de grupos... En el campo de la computación los objetos son tipos de datos y los morfismos son funciones.

A la teoría de categorías se la conoce, en broma, como *general abstract nonsense*. Parece una mera presentación taxonómica de estructuras formadas por conjuntos, operaciones y leyes, con conceptos más y más abstractos. La abstracción dificulta percibir la posible aplicación práctica. Algunos conceptos útiles son los monoides, los functores, los functores aplicativos y las mónadas. Explicados en abstracto, parecen completamente absurdos.

Tomemos, por ejemplo, la definición de functor:

Let C and D be categories. A functor F from C to D is a mapping that

- associates to each object $X \in C$ an object $F(X) \in D$,
- associates to each morphism $f : X \to Y \in C$ a morphism $F(f) : F(X) \to F(Y) \in D$ such that the following two conditions hold:
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X \in C$
  - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$.

Un functor es una transformación que preserva, en cierto sentido, la estructura. Transformar los elementos de una lista (o de un árbol) de modo que alteremos su valor pero no la secuencia y número de nodos (o las relaciones padres-hijos) es una operación frecuente en programación, y el hecho de que esa operación exista hace que la lista (o el árbol) sea un functor. Ver la lista (o el árbol) como un functor ayuda a utilizar ciertos resultados abstractos de la teoría de categorías y a conocer las limitaciones de lo que se puede hacer con ellos.

Haskell modela la clase `Functor` y ofrece una función capaz de efectuar las transformaciones propias de un functor (es la función u operador `fmap`, que "eleva" una función de un tipo a otro a una función del functor de un tipo al functor del otro tipo; como ves, *general abstract nonsense*).

Las mónadas son uno de los conceptos de la teoría de categorías que se han puesto de moda. Nos permiten, por ejemplo, trabajar en un lenguaje funcional puro con un estilo imperativo. Haskell ofrece una notación especial para tratar con mónadas y hacer más legibles los programas que las usan. La entrada/salida en Haskell, por ejemplo, es monádica.

Como dice C. D. Smith en su blog Sententia cdsmithus:

> "I predict that within the next ten years, software developers will be expected to discuss monads in the same way that most developers currently have a working vocabulary of design patterns or agile methodologies."

2.4 Programación funcional pura

Pero he de pasar (rápidamente) por las ventajas que ofrece la programación funcional **en Haskell**, que es el lenguaje funcional más puro en la zona pragmática o utilizable en el mundo real:

- **Transparencia referencial**: El resultado de una función depende única y exclusivamente de sus datos de entrada. Eso implica que no puede haber un estado que modifique el "comportamiento" de la función. Una consecuencia es que simplifica el diseño de pruebas unitarias (si es que sueles trabajar con TDD).
- **Sin efectos secundarios** (implícitos) y con **sintaxis de apoyo para uso de mónadas**: Del mismo modo que una función no depende del estado global, una función no puede modificar el estado global. Esto parece chocar con la esencia de la entrada/salida que, por naturaleza, genera efectos secundarios en el mundo o modifica los cálculos por la aparición de datos "de la nada" (es decir, no como valor que ingresa en "el sistema" como argumento de llamada a función). Pero Haskell las supera con un modelado que hace explícito, en términos monádicos, esa entrada/salida como el resultado del cálculo de un "estado del mundo" a partir de un "estado del mundo" previo.
- **Evaluación no estricta** (con técnica perezosa), sin orden de evaluación y estructuras infinitas: La evaluación perezosa permite que no se ejecute más cálculo que el necesario para alcanzar un resultado. Las estructuras infinitas plantean problemas de terminación de cálculo en lenguajes imperativos que desaparecen con la evaluación perezosa: por infinitos valores que tenga una estructura, en un cálculo finito solo emplearemos una cantidad finita de ellos y serán esos los únicos que se acaben calculando realmente.
- **Funciones currificadas, de primera clase, de orden superior**: En Haskell todo son funciones. Las funciones son valores y las funciones admiten funciones como argumentos y pueden devolver funciones como resultado. La currificación de funciones (todas tienen un solo parámetro porque las que parecen tener más de uno son, en realidad, funciones que devuelven funciones) permite expresar concisamente ciertas composiciones de funciones.
- **Tipado estático** (con types y kinds) e **inferencia de tipos**: En Haskell es frecuente decir "si compila, funciona". El tipado nos ayuda enormemente, especialmente en la refactorización de código. Además, no es muy intrusivo porque la inferencia hace innecesario declarar los tipos (salvo para resolver alguna ambigüedad por excesiva generalización). Solemos declarar tipos de funciones o variables para documentar el código.
- **Eliminación de recursión por cola**: La recursión por cola no consume memoria proporcional al número de llamadas. Ello permite montar eficientemente cálculos recursivos.
- Colección de **librerías** con potentes **abstracciones funcionales**: Hackage contienen muchas librerías que implementan algunas de las construcciones abstractas que encontrarás en los aspectos más formales y teóricos de la programación funcional. Son librerías altamente reutilizables una vez entiendes qué constructos modelan.
- **Tipos de datos algebraicos**: Los tipos de datos algebraicos permiten modelar sumas y productos de tipos de datos. Evitan mucha de la complejidad de la orientación a objetos y su aprendizaje puede recompensar, también, al usar un lenguaje orientado a objetos.
- **Concordancia de patrones** (*pattern matching*): los datos se pueden "deconstruir" sobre la marcha para definir casos en los que podemos aplicar ciertas definiciones de funciones o para seleccionar la expresión más adecuada en un contexto determinado. El resultado son programas muy expresivos y legibles, con un estilo declarativo.
- Todo el **programa** se reduce a la **evaluación** e una (mega) **expresión** para obtener un valor: ¡no hay estructuras de control de flujo porque no hay flujo que controlar!
- Librerías con **colecciones inmutables**. El diseño de programas en el que los datos no mutan fuerza a operar con estructuras de datos de modo que cada cambio genera una copia de la estructura con la modificación, a la vez que la estructura original sigue estando disponible.

Pasar a un estilo de programación sometido a todos estos cambios no es sencillo. Pero aprender efectivamente requiere salir de la zona de confort y experimentar la rareza de un mundo muy diferente al que estás habituado. La experiencia es útil tanto si decides quedarte en ese mundo raro como si, después de la excursión, decides volver a tu mundo habitual. Habrás aprendido un montón de cosas que cambiarán tu forma de ver la programación y que, seguro, tendrán un impacto positivo en tu estilo.

Bueno. Menos cháchara. La charla se titula "Por qué deberías aprender programación funcional ya mismo". Vamos a por el "ya mismo". Empezamos.

3 Ensuciémonos las manos

Aprenderemos lo básico del lenguaje con un par de katas. Veremos las cosas con cierto atropello, pero es que esto no es un tutorial paso a paso. Solo pretendo presentaros algunos elementos de Haskell y un flujo al que seguramente estáis acostumbrados si practicáis TDD.

3.1 Kata FizzBuzz

Imagine the scene. You are eleven years old, and in the five minutes before the end of the lesson, your Maths teacher decides he should make his class more "fun" by introducing a "game". He explains that he is going to point at each pupil in turn and ask them to say the next number in

sequence, starting from one. The “fun” part is that if the number is divisible by three, you instead say “fizz” and if it is divisible by five you say “buzz”. So now your maths teacher is pointing at all of your classmates in turn, and they happily shout `”one!”`, “two!”, “fizz!”, “four!”, “buzz!”... until he very deliberately points at you, fixing you with a steely gaze... time stands still, your mouth dries up, your palms become sweatier and sweatier until you finally manage to croak “fizz!”. Doom is avoided, and the pointing finger moves on.

So of course in order to avoid embarassment infront of your whole class, you have to get the full list printed out so you know what to say. Your class has about 33 pupils and he might go round three times before the bell rings for breaktime. Next maths lesson is on Thursday. Get coding!

Write a program that prints the numbers from 1 to 100. But for multiples of three print “fizz!” instead of the number and for the multiples of five print “buzz!”. For numbers which are multiples of both three and five print “fizzbuzz!”.

3.1.1 Preparar el entorno

Cada proyecto usa un conjunto de librerías. Podemos tener un repositorio local de librerías, pero es fácil que entremos en conflictos de versiones entre diferentes proyectos. Es mejor crear un repositorio local al proyecto, un sandbox. Empiezo creo un sandbox para las librerías en el directorio principal del proyecto (en nuestro caso `/Users/amarzal/Desktop/Katas/FizzBuzz/`):

```
amarzal$ cabal sandbox init
Writing a default package environment file to
/Users/amarzal/Desktop/Katas/FizzBuzz/cabal.sandbox.config
Creating a new sandbox at /Users/amarzal/Desktop/Katas/FizzBuzz/.cabal-sandbox
```

Instalamos las librerías para prueba unitaria. Tasty es un framework que permite combinar diferentes estilos de prueba unitaria. Instalamos, además, la librería HUnit con el adaptador para funcionar bajo Tasty:

```
amarzal$ cabal install tasty tasty-hunit
Resolving dependencies...
Notice: installing into a sandbox located at
/Users/amarzal/Desktop/Katas/FizzBuzz/.cabal-sandbox
Configuring ansi-terminal-0.6.2.1...
Configuring mtl-2.2.1...
Configuring stm-2.4.4...
Configuring tagged-0.8.0.1...
Configuring transformers-compat-0.4.0.4...
Configuring unbounded-delays-0.1.0.9...
Configuring text-1.2.0.4...
Building ansi-terminal-0.6.2.1...
Building tagged-0.8.0.1...
Building stm-2.4.4...
Building mtl-2.2.1...
Building transformers-compat-0.4.0.4...
Building text-1.2.0.4...
Installed transformers-compat-0.4.0.4
Building unbounded-delays-0.1.0.9...
Installed unbounded-delays-0.1.0.9
Installed stm-2.4.4
Configuring async-2.0.2...
Installed tagged-0.8.0.1
Building async-2.0.2...
Installed mtl-2.2.1
Configuring regex-base-0.93.2...
Installed ansi-terminal-0.6.2.1
Configuring ansi-wl-pprint-0.6.7.2...
Building regex-base-0.93.2...
Building ansi-wl-pprint-0.6.7.2...
Installed async-2.0.2
Installed ansi-wl-pprint-0.6.7.2
Configuring optparse-applicative-0.11.0.2...
Building optparse-applicative-0.11.0.2...
```

```
Installed regex-base-0.93.2
Installed optparse-applicative-0.11.0.2
Installed text-1.2.0.4
Configuring parsec-3.1.9...
Building parsec-3.1.9...
Installed parsec-3.1.9
Configuring regex-tdfa-rc-1.1.8.3...
Building regex-tdfa-rc-1.1.8.3...
Installed regex-tdfa-rc-1.1.8.3
Configuring tasty-0.10.1...
Building tasty-0.10.1...
Installed tasty-0.10.1
Configuring tasty-hunit-0.9.2...
Building tasty-hunit-0.9.2...
Installed tasty-hunit-0.9.2
Updating documentation index
/Users/amarzal/Desktop/Katas/FizzBuzz/.cabal-sandbox/share/doc/x86_64-osx-ghc-7.10.1/index.html
```

Cabal ha instalado todas las librerías de las que dependen tanto `tasty` como `tasty-hunit`.

3.1.2 Creación del proyecto

Ahora creamos el fichero de proyecto. Es un fichero con extensión `cabal` en el que damos datos sobre el proyecto y sus dependencias. Cabal podrá compilar, someter a pruebas e instalar el paquete en un repositorio automáticamente.

Creamos el fichero con ayuda de la propia herramienta `cabal` (aunque el resultado no es más que un fichero de texto que podríamos crear desde cero con un editor cualquiera):

```
amarzal$ cabal init
Package name? [default: FizzBuzz]
Package version? [default: 0.1.0.0]
Please choose a license:
 * 1) (none)
   2) GPL-2
   3) GPL-3
   4) LGPL-2.1
   5) LGPL-3
   6) AGPL-3
   7) BSD2
   8) BSD3
   9) MIT
  10) ISC
  11) MPL-2.0
  12) Apache-2.0
  13) PublicDomain
  14) AllRightsReserved
  15) Other (specify)
Your choice? [default: (none)]
Author name? [default: Andres Marzal]
Maintainer email? [default: amarzal AT uji.es]
Project homepage URL?
Project synopsis?
Project category:
 * 1) (none)
   2) Codec
   3) Concurrency
   4) Control
   5) Data
```

```
   6) Database
   7) Development
   8) Distribution
   9) Game
  10) Graphics
  11) Language
  12) Math
  13) Network
  14) Sound
  15) System
  16) Testing
  17) Text
  18) Web
  19) Other (specify)
Your choice? [default: (none)]
What does the package build:
   1) Library
   2) Executable
Your choice? 1
What base language is the package written in:
 * 1) Haskell2010
   2) Haskell98
   3) Other (specify)
Your choice? [default: Haskell2010]
Include documentation on what each field means (y/n)? [default: n]
Source directory:
 * 1) (none)
   2) src
   3) Other (specify)
Your choice? [default: (none)] 2

Guessing dependencies...

Generating LICENSE...
Warning: unknown license type, you must put a copy in LICENSE yourself.
Generating Setup.hs...
Generating FizzBuzz.cabal...

Warning: no synopsis given. You should edit the .cabal file and add one.
You may want to edit the .cabal file and add a Description field.
```

Se ha creado el fichero `FizzBuzz.cabal` con este contenido:

```
-- Initial FizzBuzz.cabal generated by cabal init.  For further
-- documentation, see http://haskell.org/cabal/users-guide/

name:                FizzBuzz
version:             0.1.0.0
-- synopsis:
-- description:
-- license:
license-file:        LICENSE
author:              Andres Marzal
maintainer:          amarzal AT uji.es
-- copyright:
-- category:
build-type:          Simple
-- extra-source-files:
cabal-version:       >=1.10
```

```
library
  -- exposed-modules:
  -- other-modules:
  -- other-extensions:
  build-depends:       base >=4.8 && <4.9
  hs-source-dirs:      src
  default-language:    Haskell2010
```

La marca `--` delimita el inicio de un comentario que acaba con el salto de línea. El fichero cabal no es un fichero Haskell: Cabal tiene su propia sintáxis (más similar a YAML que a Haskell), aunque copia de Haskell la sintaxis de los comentarios de una sola línea.

El directorio `src` ya lo ha creado cabal automáticamente. Ahí meteremos el código de la librería. Creamos manualmente el directorio `test` para el código con las pruebas:

```
amarzal$ mkdir test
```

Editamos el fichero `FizzBuzz.cabal` para que quede así:

```
-- Initial FizzBuzz.cabal generated by cabal init.  For further
-- documentation, see http://haskell.org/cabal/users-guide/

name:                FizzBuzz
version:             0.1.0.0
-- synopsis:
-- description:
-- license:
-- license-file:        LICENSE
author:              Andres Marzal
maintainer:          amarzal AT uji.es
-- copyright:
-- category:
build-type:          Simple
-- extra-source-files:
cabal-version:       >=1.10

library
  exposed-modules:     FizzBuzz
  -- other-modules:
  -- other-extensions:
  build-depends:       base >=4.8 && <4.9
  hs-source-dirs:      src
  default-language:    Haskell2010

test-suite test
  type:                exitcode-stdio-1.0
  hs-source-dirs:      test
  build-depends:       base
                     , tasty
                     , tasty-hunit
                     , FizzBuzz
  main-is:             Tests.hs
  default-language:    Haskell2010
```

Hemos indicado en el fichero que el módulo FizzBuzz (que definiremos en un fichero `src/FizzBuzz.hs`) es visible para quienes usen la librería (`exposed-modules`). También hemos añadido una sección nueva (`test-suite`) para las pruebas unitarias , a la que llamamos `test`, con sus propias dependencias de compilación (`build-depends`). Entre las dependencias están las librerías que hemos instalado y el paquete `FizzBuzz`, que es el resultado principal del proyecto (y al que hemos dado nombre con `name: FizzBuzz`).

3.1.3 Diseño inicial de la librería

Empezamos la kata con un diseño inicial muy basto. El diseño inicial es una función que recibe un entero y devuelve una cadena de texto. El fichero `src/FizzBuzz.hs` contendrá esa función y queda así:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n = undefined
```

Estamos creando un módulo (`module`) llamado `FizzBuzz` donde (`where`) se define una función.

La variable `fizzbuzz` es de tipo (`::`) función (`->`) y tiene como dominio los enteros (`Int`) y como codominio las cadenas (`String`). La función tiene un parámetro e, independientemente de su valor, devuelve un valor fijo: `undefined`. El valor `undefined` forma parte de cualquier tipo de datos, así que no hay conflicto de tipos. Usándolo conseguimos una compilación limpia (los tipos casan), pero aún no hemos hecho nada útil.

3.1.4 Diseño de la primera prueba

Empezamos con la primera prueba. Aplicar `fizzbuzz` al número 1 proporciona la cadena `"one!"`. Editamos el fichero `test/Tests.hs` para que quede así:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

fizzBuzzSuite :: TestTree
fizzBuzzSuite = testGroup "FizzBuzz tests"
                [ testCase "1 is one!" $ fizzBuzz 1 @?= "one!"]

main = defaultMain fizzBuzzSuite
```

Configuramos cabal:

```
amarzal$ cabal configure --enable-tests
Resolving dependencies...
Configuring FizzBuzz-0.1.0.0...
```

Las pruebas unitarias residen en su propio módulo (`Main`). El módulo empieza importando las funciones y tipos definidos en las librerías `Test.Tasty` y `Test.Tasty.HUnit`. Importa, además, el contenido de `FizzBuzz`, es decir, la función `fizzbuzz` que hemos definido en precario. A continuación crea una variable (`fizzBuzzSuite`) con el árbol de pruebas unitarias (tipo `TestTree`). Es un árbol porque la librería Tasty permite organizar jerárquicamente las pruebas unitarias como listas de listas de... Agrupamos por módulos, por funciones dentro de los módulos, por casos dentro de las funciones...

La variable `fizzBuzzSuite` se define como el resultado de aplicar la función `testGroup` (grupo de pruebas) al texto `"FizzBuzz tests"` (es el mensaje que aparecerá en pantalla cuando empiezce la ejecución de las pruebas) y a la lista (datos encerrados entre corchetes). La lista solo contiene un elemento: el que resulta de aplicar la función `testCase` al texto `"1 is one"` y a la expresión `fizzBuzz 1 @?= "one!"`. ¿Qué significa el `$`? Podríamos haber escrito la lista encerrando entre paréntesis la expresión que sigue al dólar:

```
[ testCase "1 is one!" (fizzBuzz 1 @?= "one!")]
```

La definición de `$` es esta:

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

infixr 0 $
```

El dólar es un operador de aplicación de función a argumento con un precedencia muy baja. Permite crear "barreras" entre una función y uno o más de uno de sus argumentos y ayuda a hacer más legible el código suprimiendo paréntesis.

En la expresión con la prueba aparece el operador `@?=`. Es un operador de HUnit y comprueba si el valor producido es igual al esperado. (Mnemónico: `?` es lo que producimos y `=` es lo que debe ser. Hay un operador `@=?`, por si nos gusta más el orden invertido.)

El programa acaba con la línea

```
main = defaultMain fizzBuzzSuite
```

Estamos definiendo el punto de entrada del programa (la función `main`) como una invocación a una función predefinida en `Test.Tasty` (`defaultMain`) con el argumento `fizzBuzzSuite`, es decir, nuestro conjunto de pruebas unitarias.

3.1.5 Compilación y... ¡rojo!

Construimos la librería con:

```
amarzal$ cabal build
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite 'test' for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
```

Podemos ejecutar los test con

```
amarzal$ cabal test
Preprocessing library FizzBuzz-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite 'test' for FizzBuzz-0.1.0.0...
Running 1 test suites...
Test suite test: RUNNING...
FizzBuzz tests
  1 is one!: FAIL
    Exception: Prelude.undefined

1 out of 1 tests failed (0.00s)
Test suite test: FAIL
Test suite logged to: dist/test/FizzBuzz-0.1.0.0-test.log
0 of 1 test suites (0 of 1 test cases) passed.
```

Evidentemente, no pasamos el test. Hay algún problema con mostrar el resultado en color. Invocar directamente al programa de pruebas saca los resultados en color:

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite 'test' for FizzBuzz-0.1.0.0...
FizzBuzz tests
  1 is one!: FAIL
    Exception: Prelude.undefined

1 out of 1 tests failed (0.00s)
```

Estamos en rojo.

3.1.6 Pasamos a verde

Modificamos el código fuente para pasar el test:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n = ”one!”
```

Ahora:

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.o )
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.p_o )
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite ’test’ for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [FizzBuzz changed]
Linking dist/build/test/test ...
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
FizzBuzz tests
  1 is one!: OK (0.01s)

All 1 tests passed (0.02s)
```

3.1.7 Nuevas pruebas

Pasamos a añadir una prueba sencilla más:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

fizzBuzzSuite :: TestTree
fizzBuzzSuite = testGroup ”FizzBuzz tests”
                [ testCase ”1 is one!” $ fizzbuzz 1 @?= ”one!”
                , testCase ”2 is two!” $ fizzbuzz 2 @?= ”two!”
                ]

main = defaultMain fizzBuzzSuite
```

La forma de expresar la separación elementos de una lista con comas al principio de las líneas (y no al final) es un estilo habitual en Haskell.

Evidentemente, ahora pasamos una prueba y la otra no:

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite ’test’ for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
```

```
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
FizzBuzz tests
  1 is one!: OK
  2 is two!: FAIL
    expected: "two!"
     but got: "one!"

1 out of 2 tests failed (0.01s)
```

Vamos retocar el programa. Podemos usar un `if-then-else`. La primer diferencia con respecto a un lenguaje convencional es que `if` no es una sentencia sino una expresión que, al evaluarse, proporciona un valor:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n = if n == 1 then "one!" else "two!"
```

En Haskell no hay sentencias, sino expresiones que se evalúan y proporcionan valores. Observa que no puede haber `then` sin `else` porque si la condición no es cierta, hemos de devolver algo del mismo tipo que se hubiera devuelto de ser cierta la condición. Al ejecutar `if-then-else` no busco la realización de un efecto secundario, sino el cálculo de un valor.

Podríamos definir nuestra propia función `ifThenElse` usando una construcción alternativa:

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse cond thenVal elseVal =
  case cond of
    True  -> thenVal
    False -> elseVal
```

Entonces `fizzBuzz` podría definirse así:

```
fizzbuzz n = ifThenElse (n==1) "one!" "two!"
```

Y ahora, una digresión breve sobre el hecho de que Haskell es perezoso. Esta otra función

```
fizzbuzz n = ifThenElse (n==1) ("one"++"!") ("two"++"!")
```

solo ejecutará una de las dos concatenación según el resultado de la comparación. Los argumentos no se evalúan antes de la llamada.

En cualquier caso, ya estamos en verde:

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.o )
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite 'test' for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [FizzBuzz changed]
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
FizzBuzz tests
  1 is one!: OK
  2 is two!: OK

All 2 tests passed (0.01s)
```

3.1.8 Hora de refactorizar

Vamos a hacer unas pocas refactorizaciones para aprender cosas de Haskell. Lo primero: podemos usar **guardas** para evitar el `if`:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n | n == 1 = ”one!”
fizzbuzz n | n /= 1 = ”two!”
```

La expresión que hay tras la barra | es una condición, una expresión que se evalúa a un valor de tipo `Bool` (`True` o `False`). (El operador “distinto de” es `/=`, y no `<>` o `!=`, como en otros lenguajes.)

Las dos definiciones se pueden unir en una sola:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n | n == 1 = ”one!”
           | n /= 1 = ”two!”
```

Hemos omitido el `fizzbuzz n` de la seguna línea de la definición.

Verde. Seguimos. Está claro que si la primera condición es falsa (`False`), la segunda solo puede ser cierta (`True`). Podemos reescribir la definición así:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n | n == 1 = ”one!”
           | True   = ”two!”
```

Las guardas se evalúan por el orden en el que aparecen en la definición, y solo se ejecuta la expresión asociada a la primera guarda. Por eso podemos poner `True` en la segunda guarda: si la expresión `n == 1` se evaluó a `False`, la segunda condición es cierta.

Poner `True` en la última guarda no es lo habitual. Hay un identificador que por defecto ya vale `True`:

¡

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n | n == 1     = ”one!”
           | otherwise  = ”two!”
```

Tenemos azúcar sintáctico para guardas de la forma `parámetro == valor`. Esta definición es equivalente a la anterior:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz 1 = ”one!”
fizzbuzz n = ”two!”
```

Recuerda que el orden importa. Si se usa la primera definición, no se usa la segunda. Seguimos en verde.

El parámetro `n` no se está usando en la expresión asociada a la definición. Se puede usar el símbolo _ para indicar “no me importa, no lo voy a usar” (al _ se le denomina *don’t care* o *wild card*):

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz 1 = ”one!”
fizzbuzz _ = ”two!”
```

Ahora tenemos una versión sencilla y legible.

3.1.9 Una función auxiliar

Antes de seguir, pensemos un poco. Parece claro que acabaremos imprimiendo números cualesquiera entre 1 y 100. Los números entre 1 y 19 tienen formas difícil de generalizar, pero los números entre 20 y 99 siguen un patrón: “twenty one”, “twenty two”, ..., “thirty one”, “thirty two”, ... Seguro que nos viene bien disponer de una función auxiliar que nos devuelve la transcripción de cualquier número. Bueno, en realidad, tres funciones: una que devuelva las cadenas para números entre 1 y 19, otra que devuelva la cadena correspondiente a las decenas para números entre 20 y 99, y otra que combine ambos resultados y trate también el caso del 100.

Paramos el desarrollo de `fizzbuzz` y vamos a por estas funciones. La primera se llamará `lessThan20`.

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

fizzBuzzSuite :: TestTree
fizzBuzzSuite = testGroup ”FizzBuzz tests”
                [ testGroup ”fizzbuzz” $
                  [ testCase ”1 is one!” $ fizzbuzz 1 @?= ”one!”
                  , testCase ”2 is two!” $ fizzbuzz 2 @?= ”two!”
                  ]
                , testGroup ”lessThan20” $
                  [ testCase ”1 is one” $ lessThan20 1 @?= ”one”
                  , testCase ”2 os two” $ lessThan20 2 @?= ”two”
                  ]
                ]

main = defaultMain fizzBuzzSuite
```

Hemos añadido un nuevo grupo de pruebas: las que verificarán la implementación de la función `lessThan20`. Implementemos la función `lessThan20` de menor complejidad capaz de superar las pruebas:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz 1 = ”one!”
fizzbuzz _ = ”two!”

lessThan20 :: Int -> String
lessThan20 1 = ”one”
lessThan20 2 = ”two”
```

Estamos en verde.

3.1.10 Refactorización del código de pruebas (y extensión del conjunto de casos sometidos a prueba)

Está claro que el programa de pruebas va a ser muy largo (19 pruebas, una por cada número entre 1 y 19, ambos incluidos) si vamos por este camino. Tratamos de montar las 19 pruebas con un código más compacto:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

lessThan20Answers = words (”one two three four five six seven eight nine ten ” ++
                           ”eleven twelve thirteen fourteen fifteen sixteen ” ++
                           ”seventeen eighteen nineteen”)

fizzBuzzSuite :: TestTree
fizzBuzzSuite = testGroup ”FizzBuzz tests”
                [ testGroup ”fizzbuzz” $
                  [ testCase ”1 is one!” $ fizzbuzz 1 @?= ”one!”
                  , testCase ”2 is two!” $ fizzbuzz 2 @?= ”two!”
                  ]
                , testGroup ”lessThan20” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t)
                      (zip [1..] lessThan20Answers)
                ]

main = defaultMain fizzBuzzSuite
```

La función `words`, predefinida, parte una cadena con palabras en una lista de palabras. El operador `++` concatena listas (y una cadena no es más que una lista de caracteres con una sintaxis especial). Si invocamos el intérprete interactivo de Haskell, `ghci`, podemos ilustrarlo con un par de ejemplos:

```
amarzal$ ghci
GHCi, version 7.10.1: http://www.haskell.org/ghc/  :? for help
Prelude> words ”un ejemplo con words”
[”un”,”ejemplo”,”con”,”words”]
Prelude> words ”palabra”
[”palabra”]
```

La variable `lessThan20Answers` almacena una lista con las 19 palabras. Hemos reescrito el código de las pruebas para no tener que escribir 19 lineas, una por palabra. Estudiemos este fragmento:

```
                , testGroup ”lessThan20” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t)
                      (zip [1..] lessThan20Answers)
```

Vamos por partes. La función `map` tiene este perfil:

```
map :: (a -> b) -> [a] -> [b]
```

Es decir, recibe un función de un tipo `a` a otro tipo `b` y una lista de datos de tipo `a`. El resultado es una lista de datos de tipo `b`. Veamos un ejemplo de aplicación:

```
Prelude> map (\x -> x * 2) [1,2,3]
[2,4,6]
```

En el ejemplo hemos aplicado la función anónima `\x -> x * 2` a la lista de enteros `[1,2,3]`. El resultado es la lista de enteros `[2,4,6]`.

En el fragmento estudiado:

- la función es `\(n, t) -> testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t)`

- y la lista es `zip [1..] lessThan20Answers`

Empezamos por el final. El fragmento `zip [1..] lessThan20Answers` hace una "cremallera" con los elementos de una lista *infinita* que contiene la sucesión de números naturales y la lista de palabras. La cremallera produce como resultado una lista de pares. De nuevo, un ejemplo con el intérprete interactivo ayuda a entender lo que estamos haciendo:

```
Prelude> zip [1..] [”un”,”ejemplo”,”con”,”words”]
[(1,”un”),(2,”ejemplo”),(3,”con”),(4,”words”)]
```

En el caso que nos ocupa, la lista contiene un número y su transcripción al inglés. El primer elemento de la lista será `(1,”one”)`, el segundo `(2,”two”)`... y así hasta llegar al último, que será `(19,”nineteen”)`.

La función `map` aplicará una función a cada elemento de esa lista de pares para generar una nueva lista. La función recibe un par `(n, t)`, que son un entero y una cadena, y produce como resultado:

```
testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t
```

Es decir, produce una llamada a `testCase` con la cadena que resulta de evaluar `(show n ++ ” is ” ++ t)`. La función `show` está predefinida y convierte datos de cualquier tipo (matiz: de cualquier tipo de la clase `Show`) en una cadena. El resultado de, por ejemplo, `show 1 ++ ” is ” ++ ”one”` es `”1 is one”`. A la derecha del `$` se encuentra el segundo argumento de la llamada a `testCase`, que es la comprobación de invocar `lessThan20` sobre el entero `n` produce la cadena `t`. Hemos producido una lista con 19 comprobaciones.

Pasamos a rojo porque nuestro código aún no contempla en la función `lessThan20` cada uno de los los 19 casos que se someten a prueba:

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite ’test’ for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
FizzBuzz tests
  fizzbuzz
    1 is one!:       OK
    2 is two!:       OK
  lessThan20
    1 is one:        OK
    2 is two:        OK
    3 is three:      FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    4 is four:       FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    5 is five:       FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    6 is six:        FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    7 is seven:      FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    8 is eight:      FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    9 is nine:       FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    10 is ten:       FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    11 is eleven:    FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    12 is twelve:    FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
```

```
    13 is thirteen:  FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    14 is fourteen:  FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    15 is fifteen:   FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    16 is sixteen:   FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    17 is seventeen: FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    18 is eighteen:  FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20
    19 is nineteen:  FAIL
      Exception: src/FizzBuzz.hs:(9,1)-(10,20): Non-exhaustive patterns in function lessThan20

17 out of 21 tests failed (0.02s)
```

El error advertido es que los patrones que hemos proporcionado para la definición de `lessThan20` no cubren todas las pruebas: la función no está definida en todo el dominio. Hemos de completar nuestra función `lessThan20`:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz 1 = "one!"
fizzbuzz _ = "two!"

lessThan20 :: Int -> String
lessThan20 n
  | n >0 && n < 20 =
    let answers = words ("one two three four five six seven eight nine ten " ++
                         "eleven twelve thirteen fourteen fifteen sixteen " ++
                         "seventeen eighteen nineteen")
    in answers !! (n-1)
```

La guarda de `lessThan20` hace uso del operador booleano `&&`, es decir, la conjunción lógica. Con `let ... in ...` podemos dar nombre a subexpresiones de la expresión que queremos evaluar. El operador `!!` accede a elementos de la lista por su posición (el primer elemento tiene posición o índice cero).

Ya hemos vuelto a verde:

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.o )
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite 'test' for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [FizzBuzz changed]
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
FizzBuzz tests
  fizzbuzz
    1 is one!:       OK
    2 is two!:       OK
  lessThan20
    1 is one:        OK
    2 is two:        OK
    3 is three:      OK
```

```
    4 is four:       OK
    5 is five:       OK
    6 is six:        OK
    7 is seven:      OK
    8 is eight:      OK
    9 is nine:       OK
    10 is ten:       OK
    11 is eleven:    OK
    12 is twelve:    OK
    13 is thirteen:  OK
    14 is fourteen:  OK
    15 is fifteen:   OK
    16 is sixteen:   OK
    17 is seventeen: OK
    18 is eighteen:  OK
    19 is nineteen:  OK

All 21 tests passed (0.02s)
```

Preparar una prueba para las decenas parece sencillo si copiamos el estilo de lo último que hemos hecho:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

lessThan20Answers = words (”one two three four five six seven eight nine ten ” ++
                           ”eleven twelve thirteen fourteen fifteen sixteen ” ++
                           ”seventeen eighteen nineteen”)
tensAnswers = words ”twenty thirty forty fifty sixty seventy eighty ninety”

fizzBuzzSuite :: TestTree
fizzBuzzSuite = testGroup ”FizzBuzz tests”
                [ testGroup ”fizzbuzz” $
                  [ testCase ”1 is one!” $ fizzbuzz 1 @?= ”one!”
                  , testCase ”2 is two!” $ fizzbuzz 2 @?= ”two!”
                  ]
                , testGroup ”lessThan20” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t)
                      (zip [1..] lessThan20Answers)
                , testGroup ”tens” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ tens n @?= t)
                      (zip [2..] tensAnswers)
                ]

main = defaultMain fizzBuzzSuite
```

Y la función auxiliar `tens` tampoco resulta muy diferente de `lessThan20`:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz 1 = ”one!”
fizzbuzz _ = ”two!”

lessThan20 :: Int -> String
```

```
lessThan20 n
  | n > 0 && n < 20 =
    let answers = words (”one two three four five six seven eight nine ten ” ++
                         ”eleven twelve thirteen fourteen fifteen sixteen ” ++
                         ”seventeen eighteen nineteen”)
    in answers !! (n-1)

tens :: Int -> String
tens n
  | n >= 2 && n <= 9 =
    answers !! (n-2)
    where
      answers = words  ”twenty thirty forty fifty sixty seventy eighty ninety”
```

Por ilustrar sintaxis de Haskell hemos usado una construcción alternativa al `let declaraciones in expresión`: `expresión where declaraciones`. Se puede escoger una u otro en función de lo que se entienda que mejora la legibilidad. Ahora tenemos 29 tests en verde.

3.1.11 Una función que combina a las dos anteriores

Pasamos a definir una función `number` que nos devuelve la transcripción de un número entero entre 1 y 100. No seremos exhaustivos en las pruebas. Escogeremos unos pocos casos representativos:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

lessThan20Answers = words (”one two three four five six seven eight nine ten ” ++
                           ”eleven twelve thirteen fourteen fifteen sixteen ” ++
                           ”seventeen eighteen nineteen”)
tensAnswers = words ”twenty thirty forty fifty sixty seventy eighty ninety”

fizzBuzzSuite :: TestTree
fizzBuzzSuite = testGroup ”FizzBuzz tests”
                [ testGroup ”fizzbuzz” $
                  [ testCase ”1 is one!” $ fizzbuzz 1 @?= ”one!”
                  , testCase ”2 is two!” $ fizzbuzz 2 @?= ”two!”
                  ]
                , testGroup ”lessThan20” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t)
                      (zip [1..] lessThan20Answers)
                , testGroup ”tens” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ tens n @?= t)
                      (zip [2..] tensAnswers)
                , testGroup ”number”
                  [ testCase ”1 is one” $ number 1 @?= ”one”
                  , testCase ”5 is five” $ number 5 @?= ”five”
                  , testCase ”10 is ten” $ number 10 @?= ”ten”
                  , testCase ”11 is eleven” $ number 11 @?= ”eleven”
                  , testCase ”19 is nineteen” $ number 19 @?= ”nineteen”
                  , testCase ”20 is twenty” $ number 20 @?= ”twenty”
                  , testCase ”25 is twenty five” $ number 25 @?= ”twenty five”
                  , testCase ”50 is fifty” $ number 50 @?= ”fifty”
                  , testCase ”59 is fifty nine” $ number 59 @?= ”fifty nine”
                  , testCase ”90 is ninety” $ number 90 @?= ”ninety”
                  , testCase ”91 is ninety one” $ number 91 @?= ”ninety one”
```

```
                , testCase "99 is ninety nine" $ number 99 @?= "ninety nine"
                , testCase "100 is one hundred" $ number 100 @?= "one hundred"
                ]
              ]

main = defaultMain fizzBuzzSuite
```

El código de la nueva función se apoya en las otras dos:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz 1 = "one!"
fizzbuzz _ = "two!"

lessThan20 :: Int -> String
lessThan20 n
  | n > 0 && n < 20 =
    let answers = words ("one two three four five six seven eight nine ten " ++
                         "eleven twelve thirteen fourteen fifteen sixteen " ++
                         "seventeen eighteen nineteen")
    in answers !! (n-1)

tens :: Int -> String
tens n
  | n >= 2 && n <= 9 =
    answers !! (n-2)
    where
      answers = words  "twenty thirty forty fifty sixty seventy eighty ninety"

number :: Int -> String
number n
  | 1 <= n && n < 20           = lessThan20 n
  | n `mod` 10 == 0 && n < 100 = tens (n `div` 10)
  | n < 100                    = tens (n `div` 10) ++ " " ++ lessThan20 (n `mod` 10)
  | n == 100                   = "one hundred"
```

Las guardas son relativamente complejas, pero el estilo declarativo de Haskell facilita la lectura de los diferentes casos considerados.

3.1.12 Una digresión: funciones como operadores y operadores como funciones

Las funciones `div` y `mod` se usan en estas guardas como operadores. Toda función binaria se puede usar como operador binario con los *backticks*. Mira estos ejemplos:

```
Prelude> map (\x -> x * 2) [1,2,3]
[2,4,6]
Prelude> (\x -> x * 2) `map` [1,2,3]
[2,4,6]
Prelude> div 10 3
3
Prelude> 10 `div` 3
3
Prelude> mod 10 3
1
Prelude> 10 `mod` 3
1
```

Y todo operador binario puede usarse en posición prefija, es decir, como una función. Para ello hemos de encerrar el operador entre paréntesis:

```
Prelude> 2 + 3
5
Prelude> (+) 2 3
5
Prelude> 5 ^ 3
125
Prelude> (^) 5 3
125
```

Bueno. Estamos en verde.

3.1.13 Volvemos a la función `fizzbuzz`

Hora de refactorizar la función `fizzbuzz`. Ahora es una llamada a `number` y la concatenación de `"!"` al resultado:

```
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n = number n ++ "!"

lessThan20 :: Int -> String
lessThan20 n
  | n > 0 && n < 20 =
    let answers = words ("one two three four five six seven eight nine ten " ++
                         "eleven twelve thirteen fourteen fifteen sixteen " ++
                         "seventeen eighteen nineteen")
    in answers !! (n-1)

tens :: Int -> String
tens n
  | n >= 2 && n <= 9 =
    answers !! (n-2)
    where
      answers = words  "twenty thirty forty fifty sixty seventy eighty ninety"

number :: Int -> String
number n
  | 1 <= n && n < 20           = lessThan20 n
  | n `mod` 10 == 0 && n < 100 = tens (n `div` 10)
  | n < 100                    = tens (n `div` 10) ++ " " ++ lessThan20 (n `mod` 10)
  | n == 100                   = "one hundred"
```

Seguimos en verde. Añadamos pruebas:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import FizzBuzz

lessThan20Answers = words ("one two three four five six seven eight nine ten " ++
                           "eleven twelve thirteen fourteen fifteen sixteen " ++
                           "seventeen eighteen nineteen")
tensAnswers = words "twenty thirty forty fifty sixty seventy eighty ninety"

fizzBuzzSuite :: TestTree
```

```
fizzBuzzSuite = testGroup ”FizzBuzz tests”
                [ testGroup ”fizzbuzz” $
                  [ testCase ”1 is one!” $ fizzbuzz 1 @?= ”one!”
                  , testCase ”2 is two!” $ fizzbuzz 2 @?= ”two!”
                  , testCase ”3 is fizz!” $ fizzbuzz 3 @?= ”fizz!”
                  , testCase ”4 is four!” $ fizzbuzz 4 @?= ”four!”
                  , testCase ”5 is buzz!” $ fizzbuzz 5 @?= ”buzz!”
                  , testCase ”15 is fizzbuzz!” $ fizzbuzz 15 @?= ”fizzbuzz!”
                  , testCase ”18 is fizz!” $ fizzbuzz 18 @?= ”fizz!”
                  , testCase ”22 is twenty two!” $ fizzbuzz 22 @?= ”twenty two!”
                  , testCase ”25 is buzz!” $ fizzbuzz 25 @?= ”buzz!”
                  , testCase ”60 is fizzbuzz!” $ fizzbuzz 60 @?= ”fizzbuzz!”
                  , testCase ”99 is fizz!” $ fizzbuzz 99 @?= ”fizz!”
                  , testCase ”100 is buzz!” $ fizzbuzz 100 @?= ”buzz!”
                  ]
                , testGroup ”lessThan20” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ lessThan20 n @?= t)
                      (zip [1..] lessThan20Answers)
                , testGroup ”tens” $
                  map ( \(n, t) -> testCase (show n ++ ” is ” ++ t) $ tens n @?= t)
                      (zip [2..] tensAnswers)
                , testGroup ”number”
                  [ testCase ”1 is one” $ number 1 @?= ”one”
                  , testCase ”5 is five” $ number 5 @?= ”five”
                  , testCase ”10 is ten” $ number 10 @?= ”ten”
                  , testCase ”11 is eleven” $ number 11 @?= ”eleven”
                  , testCase ”19 is nineteen” $ number 19 @?= ”nineteen”
                  , testCase ”20 is twenty” $ number 20 @?= ”twenty”
                  , testCase ”25 is twenty five” $ number 25 @?= ”twenty five”
                  , testCase ”50 is fifty” $ number 50 @?= ”fifty”
                  , testCase ”59 is fifty nine” $ number 59 @?= ”fifty nine”
                  , testCase ”90 is ninety” $ number 90 @?= ”ninety”
                  , testCase ”91 is ninety one” $ number 91 @?= ”ninety one”
                  , testCase ”99 is ninety nine” $ number 99 @?= ”ninety nine”
                  , testCase ”100 is one hundred” $ number 100 @?= ”one hundred”
                  ]
                ]

main = defaultMain fizzBuzzSuite
```

Como las pruebas también consideran casos en los que hay que devolver *fizz*, *buzz* o *fizzbuzz*, estamos en rojo (mostramos solo el inicio de la ejecución de las pruebas):

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite ’test’ for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
FizzBuzz tests
  fizzbuzz
    1 is one!:          OK
    2 is two!:          OK
    3 is fizz!:         FAIL
      expected: ”fizz!”
       but got: ”three!”
```

```
    4 is four!:         OK
    5 is buzz!:         FAIL
      expected: "buzz!"
       but got: "five!"
    15 is fizzbuzz!:    FAIL
      expected: "fizzbuzz!"
       but got: "fifteen!"
    18 is fizz!:        FAIL
      expected: "fizz!"
       but got: "eighteen!"
    22 is twenty two!:  OK
    25 is buzz!:        FAIL
      expected: "buzz!"
       but got: "twenty five!"
    60 is fizzbuzz!:    FAIL
      expected: "fizzbuzz!"
       but got: "sixty!"
    99 is fizz!:        FAIL
      expected: "fizz!"
       but got: "ninety nine!"
    100 is buzz!:       FAIL
      expected: "buzz!"
       but got: "one hundred!"
  lessThan20
    1 is one:           OK
    2 is two:           OK
...
```

Hemos de tratar los casos especiales de `fizzbuzz`: múltiplos de 3 y múltiplos de 5.

```haskell
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n
  | n `mod` 3 == 0  = "fizz!"
  | n `mod` 5 == 0  = "buzz!"
  | otherwise       = number n ++ "!"

lessThan20 :: Int -> String
lessThan20 n
  | n > 0 && n < 20 =
    let answers = words ("one two three four five six seven eight nine ten " ++
                         "eleven twelve thirteen fourteen fifteen sixteen " ++
                         "seventeen eighteen nineteen")
    in answers !! (n-1)

tens :: Int -> String
tens n
  | n >= 2 && n <= 9 =
    answers !! (n-2)
    where
      answers = words  "twenty thirty forty fifty sixty seventy eighty ninety"

number :: Int -> String
number n
  | 1 <= n && n < 20           = lessThan20 n
  | n `mod` 10 == 0 && n < 100 = tens (n `div` 10)
  | n < 100                    = tens (n `div` 10) ++ " " ++ lessThan20 (n `mod` 10)
  | n == 100                   = "one hundred"
```

Si ejecutamos la batería de pruebas, tenemos aún fallos: nos hemos dejado el caso de que el número sea múltiplo de 3 y de 5.

```
amarzal$ cabal build; dist/build/test/test
Building FizzBuzz-0.1.0.0...
Preprocessing library FizzBuzz-0.1.0.0...
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.o )
[1 of 1] Compiling FizzBuzz         ( src/FizzBuzz.hs, dist/build/FizzBuzz.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering FizzBuzz-0.1.0.0...
Preprocessing test suite 'test' for FizzBuzz-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [FizzBuzz changed]
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
FizzBuzz tests
  fizzbuzz
    1 is one!:          OK
    2 is two!:          OK
    3 is fizz!:         OK
    4 is four!:         OK
    5 is buzz!:         OK
    15 is fizzbuzz!:    FAIL
      expected: "fizzbuzz!"
       but got: "fizz!"
    18 is fizz!:        OK
    22 is twenty two!:  OK
    25 is buzz!:        OK
    60 is fizzbuzz!:    FAIL
      expected: "fizzbuzz!"
       but got: "fizz!"
    99 is fizz!:        OK
    100 is buzz!:       OK
  lessThan20
    1 is one:           OK
    2 is two:           OK
...
```

Vamos a por ese caso:

```haskell
module FizzBuzz where

fizzbuzz :: Int -> String
fizzbuzz n
  | n `mod` 3 == 0 && n `mod` 5 == 0 = "fizzbuzz!"
  | n `mod` 3 == 0                   = "fizz!"
  | n `mod` 5 == 0                   = "buzz!"
  | otherwise                        = number n ++ "!"

lessThan20 :: Int -> String
lessThan20 n
  | n > 0 && n < 20 =
    let answers = words ("one two three four five six seven eight nine ten " ++
                         "eleven twelve thirteen fourteen fifteen sixteen " ++
                         "seventeen eighteen nineteen")
    in answers !! (n-1)

tens :: Int -> String
tens n
  | n >= 2 && n <= 9 =
```

```
    answers !! (n-2)
    where
      answers = words  "twenty thirty forty fifty sixty seventy eighty ninety"

number :: Int -> String
number n
  | 1 <= n && n < 20           = lessThan20 n
  | n `mod` 10 == 0 && n < 100 = tens (n `div` 10)
  | n < 100                    = tens (n `div` 10) ++ " " ++ lessThan20 (n `mod` 10)
  | n == 100                   = "one hundred"
```

Y ya hemos vuelto a verde.

3.2 Kata Bowling

A game of ten pins bowling lasts ten frames, in each of which the bowler makes one or two attempts to knock down ten pins arranged in a triangle. If the bowler knocks down all ten pins on the first attempt (that's called a "strike"), he scores ten pins plus the number of pins knocked down on his next two rolls. If the bowler knocks down all ten pins after two attempts (that's called a "spare"), he scores ten pins plus the number of pins knocked down on his next roll. If the bowler fails to knock down all ten pins (that's called an "open frame"), he scores the number of pins he knocked down. The scores accumulate through all ten frames. At the last frame, if necessary, the pins are reset for one or two additional rolls to count the final bonus.

Example:

Frame	Pins	Kind	Bonus	Score	Total
1	1, 4	Open		$1+4=5$	$0+5=5$
2	4, 5	Open		$4+5=9$	$5+9=14$
3	6, 4	Spare	5	$6+4+5=15$	$14+15=29$
4	5, 5	Spare	10	$5+5+10=20$	$29+20=49$
5	10	Strike	0, 1	$10+0+1=11$	$49+11=60$
6	0, 1	Open		$0+1=1$	$60+1=61$
7	7, 3	Spare	6	$7+3+6=16$	$61+16=77$
8	6, 4	Spare	10	$6+4+10=20$	$77+20=97$
9	10	Strike	2, 8	$10+2+8=20$	$97+20=117$
10	2, 8	Spare	6	$2+8+6=16$	$117+16=133$

For instance, the score in the second frame is 9, the sum of the two balls in the open frame. The score in the third frame is 15, which is 10 for the spare plus 5 on the next ball. The score in the ninth frame is 20, which is 10 for the strike plus 10 for the spare on the first two balls of the tenth frame. The score in the tenth frame is 16, which is 10 for the spare plus 6 for the extra ball, which is a bonus ball not really part of any frame (the two balls of the tenth frame have already been rolled).

Your task is to write a function that calculates the score of a tenpins bowling game. When you are finished, you are welcome to read or run a suggested solution, or to post your solution or discuss the exercise in the comments below.

3.2.1 Esbozo de diseño

Empezamos con un primer diseño que, posiblemente, habrá que retocar.

Recibiremos una lista de enteros con el resultado de todas las tiradas. El juego de la tabla anterior se codificaría con

```
[1, 4, 4, 5, 6, 4, 5, 5, 10, 0, 1, 7, 3, 6, 4, 10, 2, 8, 6]
```

Una función traducirá la lista de tiradas a una lista de frames:

```
toFrames :: [Int] -> [Frame]
```

El tipo `Frame` modelará los tres tipos de frame y cada uno contendrá la información necesaria para calcular la puntuación que aporta al total. Una función nos dirá la puntuación de un frame. Otra función calculará, dada una lista de frames, la puntuación total:

```
score :: [Frame] -> Int
```

3.2.2 Infraestructura

Creamos un sandbox en un directorio `Bowling`:

```
amarzal$ cabal sandbox init
Writing a default package environment file to
/Users/amarzal/Desktop/Katas/Bowling/cabal.sandbox.config
Creating a new sandbox at /Users/amarzal/Desktop/Katas/Bowling/.cabal-sandbox
```

Instalamos las librerías de prueba unitaria en el sandbox:

```
amarzal$ cabal install tasty tasty-hunit
Resolving dependencies...
Notice: installing into a sandbox located at
/Users/amarzal/Desktop/Katas/Bowling/.cabal-sandbox
Configuring ansi-terminal-0.6.2.1...
Configuring mtl-2.2.1...
Configuring stm-2.4.4...
Configuring tagged-0.8.0.1...
Configuring transformers-compat-0.4.0.4...
Configuring unbounded-delays-0.1.0.9...
Configuring text-1.2.0.4...
Building transformers-compat-0.4.0.4...
Building stm-2.4.4...
Building text-1.2.0.4...
Building mtl-2.2.1...
Building ansi-terminal-0.6.2.1...
Building tagged-0.8.0.1...
Building unbounded-delays-0.1.0.9...
Installed transformers-compat-0.4.0.4
Installed unbounded-delays-0.1.0.9
Installed stm-2.4.4
Configuring async-2.0.2...
Installed tagged-0.8.0.1
Building async-2.0.2...
Installed mtl-2.2.1
Configuring regex-base-0.93.2...
Installed ansi-terminal-0.6.2.1
Configuring ansi-wl-pprint-0.6.7.2...
Building regex-base-0.93.2...
Building ansi-wl-pprint-0.6.7.2...
Installed async-2.0.2
Installed ansi-wl-pprint-0.6.7.2
Configuring optparse-applicative-0.11.0.2...
Building optparse-applicative-0.11.0.2...
Installed regex-base-0.93.2
Installed optparse-applicative-0.11.0.2
Installed text-1.2.0.4
Configuring parsec-3.1.9...
Building parsec-3.1.9...
Installed parsec-3.1.9
Configuring regex-tdfa-rc-1.1.8.3...
```

```
Building regex-tdfa-rc-1.1.8.3...
Installed regex-tdfa-rc-1.1.8.3
Configuring tasty-0.10.1...
Building tasty-0.10.1...
Installed tasty-0.10.1
Configuring tasty-hunit-0.9.2...
Building tasty-hunit-0.9.2...
Installed tasty-hunit-0.9.2
Updating documentation index
/Users/amarzal/Desktop/Katas/Bowling/.cabal-sandbox/share/doc/x86_64-osx-ghc-7.10.1/index.html
```

Inicializamos el proyecto creando, con ayuda, el fichero `Bowling.cabal`:

```
amarzal$ cabal init
Package name? [default: Bowling]
Package version? [default: 0.1.0.0]
Please choose a license:
 * 1) (none)
   2) GPL-2
   3) GPL-3
   4) LGPL-2.1
   5) LGPL-3
   6) AGPL-3
   7) BSD2
   8) BSD3
   9) MIT
  10) ISC
  11) MPL-2.0
  12) Apache-2.0
  13) PublicDomain
  14) AllRightsReserved
  15) Other (specify)
Your choice? [default: (none)]
Author name? [default: Andres Marzal]
1GMaintainer email? [default: amarzal AT :w
uji.es]
Project homepage URL?
Project synopsis?
Project category:
 * 1) (none)
   2) Codec
   3) Concurrency
   4) Control
   5) Data
   6) Database
   7) Development
   8) Distribution
   9) Game
  10) Graphics
  11) Language
  12) Math
  13) Network
  14) Sound
  15) System
  16) Testing
  17) Text
  18) Web
  19) Other (specify)
Your choice? [default: (none)]
```

```
What does the package build:
   1) Library
   2) Executable
Your choice? 1
What base language is the package written in:
 * 1) Haskell2010
   2) Haskell98
   3) Other (specify)
Your choice? [default: Haskell2010]
Include documentation on what each field means (y/n)? [default: n]
Source directory:
 * 1) (none)
   2) src
   3) Other (specify)
Your choice? [default: (none)] 2

Guessing dependencies...

Generating LICENSE...
Warning: unknown license type, you must put a copy in LICENSE yourself.
Generating Setup.hs...
Generating Bowling.cabal...

Warning: no synopsis given. You should edit the .cabal file and add one.
You may want to edit the .cabal file and add a Description field.
```

Creo el directorio para el código fuente de pruebas:

```
amarzal$ mkdir test
```

Y edito el fichero `Bowling.cabal` para que quede así:

```
-- Initial Bowling.cabal generated by cabal init.  For further
-- documentation, see http://haskell.org/cabal/users-guide/

name:                Bowling
version:             0.1.0.0
-- synopsis:
-- description:
-- license:
-- license-file:        LICENSE
author:              Andres Marzal
maintainer:          amarzal AT uji.es
-- copyright:
-- category:
build-type:          Simple
-- extra-source-files:
cabal-version:       >=1.10

library
  exposed-modules:     Bowling
  -- other-modules:
  -- other-extensions:
  build-depends:       base >=4.8 && <4.9
  hs-source-dirs:      src
  default-language:    Haskell2010

test-suite test
```

```
  type:               exitcode-stdio-1.0
  hs-source-dirs:     test
  build-depends:      base
                    , tasty
                    , tasty-hunit
                    , Bowling
  main-is:            Tests.hs
  default-language:   Haskell2010
```

3.2.3 Definición de un tipo de datos

Hemos de definir primero un tipo de datos que dé soporte a la representación de frames. Creamos esta primera versión de `src/Bowling.hs`:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int

toFrames :: [Int] -> [Frame]
toFrames pins = undefined
```

Estamos construyendo una infraestructura mínima y no conocemos bien Haskell, por lo que todo ha de parecer un poco críptico. `Frame` es un nuevo tipo de datos definido por el usuario. Hay tres versiones de `Frame`, cada una con su propio constructor. Cada uno de los ellos requiere dos enteros. En cada caso, el entero significa una cosa distinta.

- En `Open` cada entero es el resultado de una tirada de bolos.
- En `Spare`, el primer entero es el resultado de la primera tirada. La segunda tirada no hace falta registrarla porque sabemos que es igual a la diferencia entre 10 y la primera tirada. El segundo entero corresponde al número de bolos tumbados en la tirada que da puntuación de bonificación.
- En `Strike` sabemos que la primera tirada ha tumbado 10 bolos, así que sería redundante registrar ese número. Los dos enteros corresponden al número de bolos tirados en cada una de las tiradas de bonificación.

Podríamos haber codificado así el tipo `Frame`:

```
data Frame = Open   { pins1 :: Int
                    , pins2 :: Int
                    }
           | Spare  { pins1 :: Int
                    , bonus1 :: Int
                    }
           | Strike { bonus1 :: Int
                    , bonus2 :: Int
                    }
```

Los nombres de los campos son, en realidad funciones (en Haskell casi *todo* son funciones) y reciben el nombre de "selectores". Dado un dato de tipo `Frame`, la función `pins1 :: Frame -> Int` devolverá un entero con el número de bolos tumbados en la primera tirada. Podemos crear un dato de tipo `Frame` con, por ejemplo, `Open 1 4`. El valor de `pins1 (Open 1 4)` es `1`. Tratar de invocar `bonus1` sobre un dato construido con `Open` produce un error:

```
Prelude> pins1 (Open 1 4)
1
Prelude> bonus1 (Open 1 4)
*** Exception: No match in record selector bonus1
```

Seguiremos trabajando con la primera versión, la que no tiene selectores.

Hemos construido un tipo de datos algebraico, formado *sumando* tres definiciones, cada una de las cuales comprende datos en el *producto* cartesiano de dos enteros.

3.2.4 Primer rojo

Por el momento, de la función `toFrames` solo hemos especificado el tipo: convertirá una lista de enteros en una lista de frames. El valor de la función es `undefined` que, recuerda, representa un valor que pertenece a cualquier tipo de datos.

Vamos con las primeras pruebas unitarias. En `test/Tests.hs` escribimos:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import Bowling

bowlingSuite :: TestTree
bowlingSuite = testGroup "Bowling tests"
                [ testGroup "toFrames"
                  [ testCase "zeros are zeros" $
                      toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)
                  ]
                ]

main = defaultMain bowlingSuite
```

La estructura de este programa es muy similar al que presentamos en la kata FizzBuzz. El único elemento nuevo tiene que ver con la gestión de listas: la función `replicate n v` genera una lista de longitud `n` con el valor `v`:

```
Prelude> replicate 10 0
[0,0,0,0,0,0,0,0,0,0]
```

La primera prueba unitaria que definimos consiste en convertir una lista con 20 ceros (una partida completa en la que no hemos tumbado ningún bolo) y ver si su conversión a una lista de `Frame`s es igual a 10 frames `Open` en los que, en cada uno y para cada una de sus dos tiradas, se han tirado cero bolos.

Antes de construir el proyecto, cabal ha de configurarse:

```
amarzal$ cabal configure --enable-tests
Resolving dependencies...
Configuring Bowling-0.1.0.0...
```

Ya podemos construir el proyecto:

```
amarzal$ cabal build
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )

test/Tests.hs:13:49:
    No instance for (Eq Frame) arising from a use of '@?='
    In the second argument of '($)', namely
      'toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)'
    In the expression:
      testCase "zeros are zeros"
```

```
      $ toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)
    In the second argument of ‘testGroup’, namely
      ‘[testCase ”zeros are zeros”
        $ toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)]’
```

¡No podemos compilar!

3.2.5 Clases e instancias

El error es que estamos tratando de comparar la igualdad de una lista de `Frame` con otra lista de `Frame`, pero los objetos de tipo `Frame` no se pueden comparar entre sí. Hemos de dar una definición de igualdad entre objetos de tipo `Frame`.

Los objetos que se pueden comparar entre sí en términos de igualdad son instancias de una clase denominada `Eq`. Aquí la terminología confunde porque los términos **clase** e **instancia** tienen un significado diferente que en orientación a objetos. En Haskell usamos el término clase para referirnos a algo similar a “interfaz con, posiblemente, métodos por defecto” y usamos el término “instancia” en el sentido de “tipo de datos que implementa de la interfaz”.

El aspecto de la clase `Eq` es este (que viene predefinido):

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y  = not (x /= y)

  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Hay dos operadores y uno se define en función de otro. Esto hace que baste con definir uno de los dos para que el otro esté automáticamente definido. (Java 8 ofrece una funcionalidad similar con sus *interface default methods*.)

Vamos a hacer que `Frame` sea una instancia de la clase `Eq` definiendo manualmente el operador de igualdad (el de desigualdad quedará definido automáticamente):

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int

instance Eq Frame where
  Open x y == Open x’ y’   = x == x’ && y == y’
  Spare x y == Open x’ y’  = x == x’ && y == y’
  Strike x y == Open x’ y’ = x == x’ && y == y’
  _ == _                   = False

toFrames :: [Int] -> [Frame]
toFrames pins = undefined
```

Hemos usado *pattern matching* en la definición: si comparamos dos frames construidos con el mismo constructor, exigimos igualdad en los dos enteros que almacenan (el tipo `Int` sí es instancia de `Eq`, por lo que el operador == está ya definido entre enteros). La definición de `_ == _` cubre cualquier otro caso, y el resultado es siempre `False`.

Tratamos de compilar, pero fracasamos de nuevo:

```
amarzal$ cabal build
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option ’-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static’
```

```
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )

test/Tests.hs:13:49:
    No instance for (Show Frame) arising from a use of '@?='
    In the second argument of '($)', namely
      'toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)'
    In the expression:
      testCase "zeros are zeros"
      $ toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)
    In the second argument of 'testGroup', namely
      '[testCase "zeros are zeros"
        $ toFrames (replicate 20 0) @?= replicate 10 (Open 0 0)]'
```

Esta vez nos dice que `Frame` no es una instancia de `Show`. La clase `Show` define una función que traduce un dato del tipo que la instancia a una cadena. Si un tipo no es instancia de la clase `Show`, sus valores no pueden representarse con cadenas y, por tanto, no pueden mostrarse en pantalla. Las pruebas unitarias necesitan mostrar valores por pantalla (al mostrar los datos que producen un fallo), así que hemos de definir la clase `Frame` como instancia de la clase `Show`.

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int

instance Eq Frame where
  Open x y == Open x' y'   = x == x' && y == y'
  Spare x y == Open x' y'  = x == x' && y == y'
  Strike x y == Open x' y' = x == x' && y == y'
  _ == _                   = False

instance Show Frame where
  show (Open x y)   = "Open " ++ show x ++ " " ++ show y
  show (Spare x y)  = "Spare " ++ show x ++ " " ++ show y
  show (Strike x y) = "Strike " ++ show x ++ " " ++ show y

toFrames :: [Int] -> [Frame]
toFrames pins = undefined
```

Como la clase `Int` es instancia de `Show` (de forma predefinida), las llamadas `show x` y `show y` no producen error de compilación y generan la representación cadena de sendos enteros.

Volvamos a tratar de compilar:

```
amarzal$ cabal build
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
```

Ya compila. La especificación de instancias de `Eq` y `Show` es tan rutinaria y común que el compilador puede generarla automáticamente:

```haskell
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames pins = undefined
```

La línea `deriving (Eq, Show)` al final de la declaración del tipo `Frame` indica al compilador que debe generar el código necesario para que ese tipo sea instancia de esas dos clases. El compilador sabe generar código para generar las instancias más básicas:

- `Eq`: igualdad/desigualdad
- `Show`: conversión a cadena vía la función `show`
- `Ord`: comparación mediante operadores <, <=, >, >= y funciones `min`, `max` y `compare`. Basta con definir la función compare, que tiene perfil `compare :: a -> a -> Ordering`, donde `Ordering` es un tipo con tres valores: `LT`, `GT`, `EQ`.
- `Bounded`: tipos con valor mínimo (`minBound`) y màximo (`maxBound`).
- `Read`: define una función `read` que permite obtener un valor a partir de una cadena (sentido inverso a `show`).

Compilamos esta versión más simple:

```
amarzal$ cabal build
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [Bowling changed]
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
```

Vamos con la primera ejecución de pruebas, que ha de fallar:

```
amarzal$ dist/build/test/test
Bowling tests
  toFrames
    zeros are zeros: FAIL
      Exception: Prelude.undefined

1 out of 1 tests failed (0.01s)
```

3.2.6 A por el primer verde

Diseñemos ahora una definición de `toFrames` que permita superar el primer rojo:

```haskell
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames pins = replicate 10 (Open 0 0)
```

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are zeros: OK

All 1 tests passed (0.01s)
```

3.2.7 Pruebas más completas

Creamos varias pruebas con partidas en las que todos los frames son opens:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import Bowling

tests :: [(String, [Int], [Frame])]
tests =
  [ ("zeros are open 0 0"
    , replicate 20 0
    , replicate 10 (Open 0 0)
    )
  , ("ones are open 1 1"
    , replicate 20 1
    , replicate 10 (Open 1 1)
    )
  , ("4+5s are open 4 5"
    , take 20 $ cycle [4, 5]
    , replicate 10 (Open 4 5)
    )
  ]

bowlingSuite :: TestTree
bowlingSuite = testGroup "Bowling tests"
                [ testGroup "toFrames" $
                  map (\(label, input, expected) ->
                          testCase label $ toFrames input @?= expected) tests
                ]

main = defaultMain bowlingSuite
```

La variable `tests` es una lista de tripletas, cada una de las cuales almacena una cadena, una lista de enteros y una lista de frames. En el árbol de pruebas recorremos esa lista con `map` y generamos un `testCase` para cada tripleta.

Ejecutamos las pruebas:

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
```

```
Preprocessing library Bowling-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0: OK
    ones are open 1 1:  FAIL
      expected: [Open 1 1,Open 1 1,Open 1 1,Open 1 1,Open 1 1,Open 1 1,Open 1 1,Open 1 1,Open 1 1,Open 1 1]
       but got: [Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0]
    4+5s are open 4 5:  FAIL
      expected: [Open 4 5,Open 4 5,Open 4 5,Open 4 5,Open 4 5,Open 4 5,Open 4 5,Open 4 5,Open 4 5,Open 4 5]
       but got: [Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0,Open 0 0]

2 out of 3 tests failed (0.01s)
```

Vamos a por una primera definición válida de `toFrames`:

```haskell
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames (x:y:ys) = Open x y : toFrames ys
toFrames []       = []
```

La función se define para dos casos. El segundo es el más sencillo: la aplicación a una lista vacía devuelve una lista vacía. El primer caso utiliza *pattern matching* sobre listas. El patrón `x:y:ys` se lee: "dada una lista con al menos dos elementos, a los que llamaremos `x` e `y` y a cuyos restantes elementos llamaremos `ys`". Las listas que hemos visto hasta ahora se construyen separando con comas valores encerrados entre corchetes, pero hay una notación alternativa. La lista `[1,2,3]` se puede representar como `1:2:3:[]`. La segunda representación refleja mejor el proceso constructivo: los dos puntos son un operador que construye la lista a partir de un valor y otra lista. El perfil de ese operador es `(:) :: a -> [a] -> [a]`. Es asociativo por la derecha, así que `1:2:3:[]` se equivale a `1:(2:(3:[]))`.

Estos ejemplos ayudarán a interpretar algunos patrones sobre listas:

```haskell
-- Ejemplos de patrones sobre listas
f :: [a] -> String
f xs       = "Una lista, no importa su número de elementos. xs representa a toda la lista."
f (x:xs)   = "una lista con al menos un elemento. x es ese elemento y xs es el resto de elementos."
f (x:y:xs) = "una lista con al menos dos elementos. x e y son los dos primeros elementos y xs es el resto."
f (x:[])   = "una lista con un solo elemento, al que denominamos x."
f []       = "la lista vacía."
f [x]      = "una lista con un solo elemento, al que denominamos x."
f [x, y]   = "una lista con dos elementos, a los que denominamos x e y."
f [x, _]   = "una lista con dos elementos. El primero es x, el segundo no me importa."
f (_:xs)   = "una lista con al menos un elemento. El primero no me importa, el resto es xs."
f (x:_)    = "una lista con al menos un elemento. El primero es x, el resto no me importa."
```

Ya estamos en verde:

```
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [Bowling changed]
```

```
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0: OK
    ones are open 1 1:  OK
    4+5s are open 4 5:  OK

All 3 tests passed (0.01s)
```

3.2.8 Partidas con spares

Vamos a modelar partidas que pueden tener frames de tipo spare. Empezamos por definir las pruebas:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import Bowling

tests :: [(String, [Int], [Frame])]
tests =
  [ ("zeros are open 0 0"
    , replicate 20 0
    , replicate 10 (Open 0 0)
    )
  , ("ones are open 1 1"
    , replicate 20 1
    , replicate 10 (Open 1 1)
    )
  , ("4+5s are open 4 5"
    , take 20 $ cycle [4, 5]
    , replicate 10 (Open 4 5)
    )
  , ("spares in non last position"
    , let spare = [1, 9]
          opens n = take (2*n) $ cycle [3, 3]
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    , let spare = [Spare 1 3]
          opens n = replicate n (Open 3 3)
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    )
  , ("spare in last position"
    , take 18 (cycle [0, 0]) ++ [1, 9, 5]
    , replicate 9 (Open 0 0) ++ [Spare 1 5]
    )
    ,
  ]

bowlingSuite :: TestTree
bowlingSuite = testGroup "Bowling tests"
                [ testGroup "toFrames" $
                  map (\(label, input, expected) ->
                          testCase label $ toFrames input @?= expected) tests
                ]

main = defaultMain bowlingSuite
```

Compila, pero hay fallos en los test:

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0:          OK
    ones are open 1 1:           OK
    4+5s are open 4 5:           OK
    spares in non last position: FAIL
      expected: [Spare 1 3,Open 3 3,Open 3 3,Spare 1 3,Open 3 3,Open 3 3,Open 3 3,Open 3 3,Spare 1 3,Open 3 3]
       but got: [Open 1 9,Open 3 3,Open 3 3,Open 1 9,Open 3 3,Open 3 3,Open 3 3,Open 3 3,Open 1 9,Open 3 3]
    spare in last position:      FAIL
      message threw an exception: src/Bowling.hs:(9,1)-(10,22): Non-exhaustive patterns in function toFrames

2 out of 5 tests failed (0.01s)
```

Vamos a modificar `toFrames` para superar el primero de los dos fallos. Parece que necesitamos detectar los spares y crear el `Frame` apropiado:

```haskell
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames (x:y:z:ys)
  | x+y == 10 = Spare x z : toFrames (z:ys)
  | otherwise = Open x y : toFrames (z:ys)
toFrames []   = []
```

La primera línea definitoria de `toFrames` considera listas con al menos tres elementos, pero distingue dos casos:

- que la suma de las dos primeras tiradas sea 10, en cuyo caso estamos ante un `Spare` cuya puntuación resultará de sumar a 10 el valor de la siguiente tirada;
- que no lo sea, en cuyo estamos ante un `Open`.

Ejecutamos las pruebas:

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
Linking dist/build/test/test ...
```

```
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0:          FAIL
      Exception: src/Bowling.hs:(9,1)-(12,20): Non-exhaustive patterns in function toFrames
    ones are open 1 1:           FAIL
      Exception: src/Bowling.hs:(9,1)-(12,20): Non-exhaustive patterns in function toFrames
    4+5s are open 4 5:           FAIL
      Exception: src/Bowling.hs:(9,1)-(12,20): Non-exhaustive patterns in function toFrames
    spares in non last position: FAIL
      message threw an exception: src/Bowling.hs:(9,1)-(12,20): Non-exhaustive patterns in function toFrames
    spare in last position:      FAIL
      message threw an exception: src/Bowling.hs:(9,1)-(12,20): Non-exhaustive patterns in function toFrames

5 out of 5 tests failed (0.05s)
```

Hemos desandado el camino. La definición que hemos dado genera problemas porque, como hemos dicho, se espera que las listas tengan

- tres o más elementos (patrón `(x:y:z:ys)`)
- o ningún elemento (patrón `[]`).

Las listas con uno o dos elementos no se modelan con ningún patrón en la definición de `toFrames`. Vamos a darles el tratamiento adecuado:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames (x:y:z:ys)
  | x+y == 10   = Spare x z : toFrames (z:ys)
  | otherwise   = Open x y : toFrames (z:ys)
toFrames [x, y] = [Open x y]
toFrames []     = []
```

Ejecutamos de nuevo las pruebas:

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0:          OK
    ones are open 1 1:           OK
    4+5s are open 4 5:           OK
    spares in non last position: OK
    spare in last position:      FAIL
      Exception: src/Bowling.hs:(9,1)-(13,20): Non-exhaustive patterns in function toFrames

1 out of 5 tests failed (0.01s)
```

Ya solo falla una prueba unitaria.

Nos falta el caso especial del spare en última posición. El final de lista con tres elementos cuyos dos primeros suman 10 genera una llamada con una lista que solo tiene un elemento. Ese caso no está cubierto por ningún patrón.

Como esa lista solo tiene sentido en el frame que ocupa la última posición, nos vendría bien controlar el número de frame en el que nos encontramos para detectar cuándo estamos en esa situación. Rediseñamos el cuerpo de `toFrames` para llevar ese control:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames pins = go 1 pins
  where
    go 10 [x, y]    = [Open x y]
    go 10 [x, y, z]
      | x + y == 10 = [Spare x z]
    go n (x:y:z:ys)
      | x + y == 10 = Spare x z : go (n+1) (z:ys)
      | otherwise   = Open x y  : go (n+1) (z:ys)
```

La función `toFrames` empieza invocando a una función auxiliar `go` definida tras el `where` (el nombre es un convenio habitual: la función auxiliar suele recibir ese nombre). La función `go` lleva el control del frame en que nos encontramos con una variable `n`. Los casos en que `n` vale 10 son los que corresponden a tiradas finales. Solo puede haber una juego final válido con 2 o con 3 tiradas. El primer caso corresponde a un open y el segundo a un spare.

Ejecutemos las pruebas, aunque empieza a resultar evidente, viendo qué casos cubren las guardas, que nuestro código tienen “agujeros”, casos que escapan a nuestro control. Luego nos ocuparemos de ellos.

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o ) [Bowling changed]
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0:          OK
    ones are open 1 1:           OK
    4+5s are open 4 5:           OK
    spares in non last position: OK
    spare in last position:      OK

All 5 tests passed (0.01s)
```

Estamos en verde, aunque tendremos que ampliar la batería de pruebas más adelante.

3.2.9 Partidas con strikes

Vamos con un primer tratamiento de los strikes:

```haskell
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import Bowling

tests :: [(String, [Int], [Frame])]
tests =
  [ ("zeros are open 0 0"
    , replicate 20 0
    , replicate 10 (Open 0 0)
    )
  , ("ones are open 1 1"
    , replicate 20 1
    , replicate 10 (Open 1 1)
    )
  , ("4+5s are open 4 5"
    , take 20 $ cycle [4, 5]
    , replicate 10 (Open 4 5)
    )
  , ("spares in non last position"
    , let spare = [1, 9]
          opens n = take (2*n) $ cycle [3, 3]
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    , let spare = [Spare 1 3]
          opens n = replicate n (Open 3 3)
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    )
  , ("spare in last position"
    , take 18 (cycle [3, 3]) ++ [1, 9, 5]
    , replicate 9 (Open 3 3) ++ [Spare 1 5]
    )
  , ("strike in non last position"
    , let strike = [10]
          opens n = take (2*n) $ cycle [3, 3]
      in strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    , let strike = [Strike 3 3]
          opens n = replicate n (Open 3 3)
      in strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    )
  , ("strike in last position"
    , take 18 (cycle [3, 3]) ++ [10, 5, 5]
    , replicate 9 (Open 3 3) ++ [Strike 5 5]
    )
  ]

bowlingSuite :: TestTree
bowlingSuite = testGroup "Bowling tests"
                [ testGroup "toFrames" $
                  map (\(label, input, expected) ->
                          testCase label $ toFrames input @?= expected) tests
                ]

main = defaultMain bowlingSuite
```

Ejecutamos pruebas:

```
amarzal$ cabal build; dist/build/test/test
```

```
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
Bowling tests
  toFrames
    zeros are open 0 0:          OK
    ones are open 1 1:           OK
    4+5s are open 4 5:           OK
    spares in non last position: OK
    spare in last position:      OK
    strike in non last position: FAIL
      message threw an exception: src/Bowling.hs:(11,5)-(16,49): Non-exhaustive patterns in function go
    strike in last position:     FAIL
      message threw an exception: src/Bowling.hs:(11,5)-(16,49): Non-exhaustive patterns in function go

2 out of 7 tests failed (0.00s)
```

Evidentemente, estamos en rojo: nuestro programa aún no cubre ningún caso de strike. Modificamos el programa en la línea de lo que hemos hecho para tratar la detección de spares:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> [Frame]
toFrames pins = go 1 pins
  where
    go 10 [x, y]    = [Open x y]
    go 10 [x, y, z]
      | x     == 10 = [Strike y z]
      | x + y == 10 = [Spare x z]
    go n (x:y:z:ys)
      | x     == 10 = Strike y z : go (n+1) (y:z:ys)
      | x + y == 10 = Spare x z : go (n+1) (z:ys)
      | otherwise   = Open x y  : go (n+1) (z:ys)
```

Y pasamos a verde:

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
Preprocessing library Bowling-0.1.0.0...
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.o )
[1 of 1] Compiling Bowling          ( src/Bowling.hs, dist/build/Bowling.p_o )
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0:          OK
    ones are open 1 1:           OK
    4+5s are open 4 5:           OK
```

```
    spares in non last position: OK
    spare in last position:      OK
    strike in non last position: OK
    strike in last position:     OK

All 7 tests passed (0.01s)
```

3.2.10 Arreglando el *code smell*

Como ya hemos apuntado antes, una inspección del código hace sospechar que hay situaciones no contempladas. Por ejemplo, hay casos en los que nos preguntamos si `x` vale 10 o si `x+y` vale 10, pero no tenemos tratamiento para el caso en el que no ocurra ni una cosa ni otra. O, si la décima trama tiene dos tiradas, decidimos que es un open frame, suponiendo que no se dan las circunstancias para un spare o strike. Y es que, si se dieran las circunstancias no sabríamos qué hacer por la sencilla razón de que la secuencia de entrada sería incorrecta. Esto nos enfrenta a la necesidad de dar un tratamiento para las entradas incorrectas. Toca rediseñar. La función `toFrames` pasa a tener este perfil:

```
toFrames :: [Int] -> Maybe [Frame]
```

El tipo `Maybe` está predefinido en Haskell así:

```
data Maybe a = Just a
             | Nothing
```

Estos son algunos ejemplos de datos de tipo `Maybe Int`: `Just 1`, `Just 10`, `Nothing`.

Con ese tipo de datos modelamos la posibilidad de que una función no compute un valor por cualquier causa. Si el cálculo no es problemático, la respuesta aparece "envuelta" en un `Just`, y si lo es, codificamos este extremo con un `Nothing`.

Adaptamos las pruebas unitarias:

```
port Test.Tasty
import Test.Tasty.HUnit

import Bowling

tests :: [(String, [Int], Maybe [Frame])]
tests =
  [ ("zeros are open 0 0"
    , replicate 20 0
    , Just $ replicate 10 (Open 0 0)
    )
  , ("ones are open 1 1"
    , replicate 20 1
    , Just $ replicate 10 (Open 1 1)
    )
  , ("4+5s are open 4 5"
    , take 20 $ cycle [4, 5]
    , Just $ replicate 10 (Open 4 5)
    )
  , ("spares in non last position"
    , let spare = [1, 9]
          opens n = take (2*n) $ cycle [3, 3]
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    , let spare = [Spare 1 3]
          opens n = replicate n (Open 3 3)
      in Just $ spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    )
  , ("spare in last position"
    , take 18 (cycle [3, 3]) ++ [1, 9, 5]
```

```
    , Just $ replicate 9 (Open 3 3) ++ [Spare 1 5]
    )
  , ("strike in non last position"
    , let strike = [10]
          opens n = take (2*n) $ cycle [3, 3]
      in strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    , let strike = [Strike 3 3]
          opens n = replicate n (Open 3 3)
      in Just $ strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    )
  , ("strike in last position"
    , take 18 (cycle [3, 3]) ++ [10, 5, 5]
    , Just $ replicate 9 (Open 3 3) ++ [Strike 5 5]
    )
  ]

bowlingSuite :: TestTree
bowlingSuite = testGroup "Bowling tests"
                [ testGroup "toFrames" $
                  map (\(label, input, expected) ->
                          testCase label $ toFrames input @?= expected) tests
                ]

main = defaultMain bowlingSuite
```

Se complica un poco el código. Observa que usamos `$` para ahorrar algunos paréntesis A la izquierda del dólar está el constructor `Just` y `$` es un operador de aplicación de función. Y es que `Just` es, también, una función. Recuerda que en Haskell casi todo son funciones.

Hacemos el cambio mínimo que nos permite seguir en verde:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> Maybe [Frame]
toFrames pins = Just $ go 1 pins
  where
    go 10 [x, y]    = [Open x y]
    go 10 [x, y, z]
      | x     == 10 = [Strike y z]
      | x + y == 10 = [Spare x z]
    go n (x:y:z:ys)
      | x     == 10 = Strike y z : go (n+1) (y:z:ys)
      | x + y == 10 = Spare x z : go (n+1) (z:ys)
      | otherwise   = Open x y  : go (n+1) (z:ys)
```

Seguimos en verde, pero lo único que hemos hecho de momento es envolver el mismo resultado que teníamos antes en un `Maybe` que siempre se construye con `Just`. Es decir, aún no estamos haciendo un tratamiento de errores, aunque tenemos el andamio preparado.

Añadimos a las pruebas una lista de enteros que no corresponde a una partida y el programa sigue fallando:

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit
```

```
import Bowling

tests :: [(String, [Int], Maybe [Frame])]
tests =
  [ ("zeros are open 0 0"
    , replicate 20 0
    , Just $ replicate 10 (Open 0 0)
    )
  , ("ones are open 1 1"
    , replicate 20 1
    , Just $ replicate 10 (Open 1 1)
    )
  , ("4+5s are open 4 5"
    , take 20 $ cycle [4, 5]
    , Just $ replicate 10 (Open 4 5)
    )
  , ("spares in non last position"
    , let spare = [1, 9]
          opens n = take (2*n) $ cycle [3, 3]
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    , let spare = [Spare 1 3]
          opens n = replicate n (Open 3 3)
      in Just $ spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    )
  , ("spare in last position"
    , take 18 (cycle [3, 3]) ++ [1, 9, 5]
    , Just $ replicate 9 (Open 3 3) ++ [Spare 1 5]
    )
  , ("strike in non last position"
    , let strike = [10]
          opens n = take (2*n) $ cycle [3, 3]
      in strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    , let strike = [Strike 3 3]
          opens n = replicate n (Open 3 3)
      in Just $ strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    )
  , ("strike in last position"
    , take 18 (cycle [3, 3]) ++ [10, 5, 5]
    , Just $ replicate 9 (Open 3 3) ++ [Strike 5 5]
    )
  , ("ill formed play"
    , [0, 1]
    , Nothing
    )
  ]

bowlingSuite :: TestTree
bowlingSuite = testGroup "Bowling tests"
               [ testGroup "toFrames" $
                 map (\(label, input, expected) ->
                         testCase label $ toFrames input @?= expected) tests
               ]

main = defaultMain bowlingSuite
```

Ejecutamos las pruebas:

```
amarzal$ cabal build; dist/build/test/test
Building Bowling-0.1.0.0...
```

```
Preprocessing library Bowling-0.1.0.0...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
In-place registering Bowling-0.1.0.0...
Preprocessing test suite 'test' for Bowling-0.1.0.0...
[1 of 1] Compiling Main             ( test/Tests.hs, dist/build/test/test-tmp/Main.o )
Linking dist/build/test/test ...
ld: warning: directory not found for option '-L/private/tmp/ghc20150402-41611-1b15z9c/ghc-7.10.1/gmp-static'
Bowling tests
  toFrames
    zeros are open 0 0:          OK
    ones are open 1 1:           OK
    4+5s are open 4 5:           OK
    spares in non last position: OK
    spare in last position:      OK
    strike in non last position: OK
    strike in last position:     OK
    ill formed play:             FAIL
      message threw an exception: src/Bowling.hs:(11,5)-(18,49): Non-exhaustive patterns in function go

1 out of 8 tests failed (0.01s)
```

Las pruebas unitarias nos han permitido detectar un problema de completitud en nuestra función `go`. La función auxiliar `go` debería detectar errores tan pronto pueda y a abortar la construcción de la lista inmediatamente:

```haskell
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> Maybe [Frame]
toFrames pins = go 1 pins
  where
    go 10 [x, y]
      | x + y < 10 = Just [Open x y]
      | otherwise  = Nothing
    go 10 [x, y, z]
      | x     == 10 = Just [Strike y z]
      | x + y == 10 = Just [Spare x z]
      | otherwise   = Nothing
    go n (x:y:z:ys)
      | x     == 10 = fmap (Strike y z :) $ go (n+1) (y:z:ys)
      | x + y == 10 = fmap (Spare x z :) $ go (n+1) (z:ys)
      | x + y < 10  = fmap (Open x y  :) $ go (n+1) (z:ys)
      | otherwise   = Nothing
    go _ _ = Nothing
```

Los casos no contemplados por los patrones y guardas que ya teníamos corresponden a situaciones de error.

3.2.11 Una digresión: functores

La construcción de la lista resulta compleja si no se está acostumbrado a los patrones de Haskell. Analicemos la expresión que dice

```haskell
fmap (Strike y z :) $ go (n+1) (y:z:ys)
```

¿Qué es `fmap`? `fmap` es lo mismo que `map`, pero para cualquier instancia de `Functor`. Las listas son un caso particular de functor, pero hay muchos más. Entre ellos se cuenta `Maybe`. Vamos a ver la definición de la clase `Functor`:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

La clase exige definir una función `fmap` que recibe una función de tipo `a -> b` y un objeto del tipo `f a` para devolver un objeto del tipo `f b`. ¿Qué es un objeto del tipo `f a`? Pues depende, naturalmente, de `f`. Hemos dicho antes que las listas son instancias de `Functor`. Vemos cómo:

```haskell
instance Functor [] where
  fmap = map
```

Hemos hecho un poco de trampa al recurrir a `map` (pero es así como se define `fmap` en el módulo `Data.Functor`). Esta versión no hace uso de `map`:

```haskell
instance Functor [] where
  fmap fun (x:xs) = fun x : fmap fun xs
  fmap _ [] = []
```

Y aún hay otra definición más clara que hace uso de las listas comprensivas:

```haskell
instance Functor [] where
  fmap fun xs = [fun x | x <- xs]
```

`Maybe` también es una instancia de `Functor`:

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

Estamos aprovechando que `Maybe` es un *functor* para mapear una función que añade a la cabeza de la lista contenida en un `Maybe` (si la hay) un elemento nuevo. Si no hay lista, es decir, si en lugar de `Just lista` tenemos un `Nothing`, la inserción no tiene efecto y obtenemos un `Nothing`, propagando el resultado que advierte del fallo.

Aún no hemos acabado de analizar esta línea:

```haskell
fmap (Strike y z :) $ go (n+1) (y:z:ys)
```

La función `fmap` aplica una función `(Strike y z :)` a un dato de tipo `Maybe [Frame]`. El segundo argumento es `go (n+1) (y:z:ys)` y el resultado es, o bien un `Just` con una lista, o bien un `Nothing`. Lo que cuesta de ver es que `(Strike y z :)` sea una función. Lo es. Para entender el porqué hemos de hablar primero de secciones.

3.2.12 Otra digresión: secciones

Sabemos que los operadores binarios Haskell se pueden usar de dos modos:

- En posición infija: `2 + 4`
- En posición prefija: `(+) 2 4`

Hay otra forma de usarlos: como secciones. La expresión `(2+)` es una función que significa "sumar dos con lo que valga el argumento". Mira cómo se usa:

- `(2+) 4`

Equivale, por tanto a `\x -> 2+x`.

Y la expresión `(+4)` significa "suma lo que valga el argumento con 4". Se usa así:

- `(+4) 2`

El operador dos puntos es el constructor de listas. La expresión `(1:)` significa "construye una lista cuyo primer elemento es 1 y cuyo resto es la lista que te suministren como argumento". He aquí un ejemplo de uso:

- (1:) [2, 3]

Ahora ya podemos entender la función `(Strike y z :)`. Significa "función que añade por la cabeza el valor `Strike y z` a una lista". La lista está en un `Maybe`, así que hemos necesitado a `fmap` para "entrar" en el interior del `Maybe`:

```
fmap (Strike y z :) $ go (n+1) (y:z:ys)
```

Decimos que `fmap` "eleva" (*lifts*) una función del mundo normal al mundo del functor (en este caso, del functor `Maybe`).

El valor devuelto por `go (n+1) (y:z:ys)` es de tipo `Maybe [Frame]`, así que puede ser una lsta envuelta en un `Just` o `Nothing`. En el primer caso, `fmap` "inyectará" dentro del `Just` la invocación a la función `(Strike y z :)`, añadiendo a la lista un elemento por la cabeza. En el segundo caso no se invocará esa función, porque `Nothing` no tiene contenido alguno al que aplicar nada.

3.2.13 Una función que calcula la puntuación de un frame

Podríamos preparar más pruebas unitarias para detectar otras partidas de bolos mal construidas (frames cuyas tiradas suman más que 10, valores negativos en la lista, listas con más tiradas que las necesarias para formar diez frames, etcétera), pero vamos a diseñar primero la función `score`.

Para ello nos vendrá bien una función `framesScore :: Frame -> Int` que nos proporcione la puntuación de un `Frame`:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> Maybe [Frame]
toFrames pins = go 1 pins
  where
    go 10 [x, y]
      | x + y < 10 = Just [Open x y]
      | otherwise  = Nothing
    go 10 [x, y, z]
      | x     == 10 = Just [Strike y z]
      | x + y == 10 = Just [Spare x z]
      | otherwise   = Nothing
    go n (x:y:z:ys)
      | x     == 10 = fmap (Strike y z :) $ go (n+1) (y:z:ys)
      | x + y == 10 = fmap (Spare x z :) $ go (n+1) (z:ys)
      | x + y < 10  = fmap (Open x y  :) $ go (n+1) (z:ys)
      | otherwise   = Nothing
    go _ _ = Nothing

frameScore :: Frame -> Int
frameScore (Open x y)   = x + y
frameScore (Spare _ y)  = 10 + y
frameScore (Strike x y) = 10 + x + y
```

La función `frameScore` es tan trivial que no prepararemos pruebas unitarias. Si la función de puntuación de una partida, `score`, supera las pruebas unitarias consideraremos que hay garantía suficiente de que `frameScore` está bien.

3.2.14 Función de puntuación de un partida

Antes de definir la función `score` creamos una prueba unitaria con una partida cuya puntuación conocemos: la del ejemplo de la especificación.

```
module Main where

import Test.Tasty
import Test.Tasty.HUnit

import Bowling

tests :: [(String, [Int], Maybe [Frame])]
tests =
  [ (”zeros are open 0 0”
    , replicate 20 0
    , Just $ replicate 10 (Open 0 0)
    )
  , (”ones are open 1 1”
    , replicate 20 1
    , Just $ replicate 10 (Open 1 1)
    )
  , (”4+5s are open 4 5”
    , take 20 $ cycle [4, 5]
    , Just $ replicate 10 (Open 4 5)
    )
  , (”spares in non last position”
    , let spare = [1, 9]
          opens n = take (2*n) $ cycle [3, 3]
      in spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    , let spare = [Spare 1 3]
          opens n = replicate n (Open 3 3)
      in Just $ spare ++ opens 2 ++ spare ++ opens 4 ++ spare ++ opens 1
    )
  , (”spare in last position”
    , take 18 (cycle [3, 3]) ++ [1, 9, 5]
    , Just $ replicate 9 (Open 3 3) ++ [Spare 1 5]
    )
  , (”strike in non last position”
    , let strike = [10]
          opens n = take (2*n) $ cycle [3, 3]
      in strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    , let strike = [Strike 3 3]
          opens n = replicate n (Open 3 3)
      in Just $ strike ++ opens 2 ++ strike ++ opens 4 ++ strike ++ opens 1
    )
  , (”strike in last position”
    , take 18 (cycle [3, 3]) ++ [10, 5, 5]
    , Just $ replicate 9 (Open 3 3) ++ [Strike 5 5]
    )
  , (”ill formed play”
    , [0, 1]
    , Nothing
    )
  ]

bowlingSuite :: TestTree
bowlingSuite = testGroup ”Bowling tests”
                [ testGroup ”toFrames” $
                  map (\(label, input, expected) ->
```

```
                    testCase label $ toFrames input @?= expected) tests
            , testGroup "score"
                [ let Just frames =
                        toFrames [1, 4, 4, 5, 6, 4, 5, 5, 10, 0, 1, 7, 3, 6, 4, 10, 2, 8, 6]
                  in testCase "spec example" $ score frames @?= 133
                ]
            ]

main = defaultMain bowlingSuite
```

El código pasa a ser:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> Maybe [Frame]
toFrames pins = go 1 pins
  where
    go 10 [x, y]
      | x + y < 10 = Just [Open x y]
      | otherwise  = Nothing
    go 10 [x, y, z]
      | x     == 10 = Just [Strike y z]
      | x + y == 10 = Just [Spare x z]
      | otherwise   = Nothing
    go n (x:y:z:ys)
      | x     == 10 = fmap (Strike y z :) $ go (n+1) (y:z:ys)
      | x + y == 10 = fmap (Spare x z :) $ go (n+1) (z:ys)
      | x + y < 10  = fmap (Open x y  :) $ go (n+1) (z:ys)
      | otherwise   = Nothing
    go _ _ = Nothing

frameScore :: Frame -> Int
frameScore (Open x y)   = x + y
frameScore (Spare _ y)  = 10 + y
frameScore (Strike x y) = 10 + x + y

score :: [Frame] -> Int
score frames = sum $ map frameScore frames
```

La definición de `score` es directa, pero explicarla requiere que tomemos otro desvío.

3.2.15 Otra digresión: map-reduce

Nótese que codificamos sin bucles, haciendo uso de `map`. Disponemos de una función sumatorio (`sum`) predefinida, pero no hubiera costado mucho diseñar nuestro propio sumatorio como una reducción de la lista:

```
sumatorio :: [Int] -> Int
sumatorio (x:xs) = x + sumatorio xs
sumatorio [] = 0
```

La reducción de una lista a un solo valor es una operación tan corriente, que dispone de herramientas ad-hoc:

```
sumatorio xs = foldl (+) 0 xs
```

La función `foldl` es un reductor de listas. Con reducciones podemos calcular el productorio, el minimo, el máximo, encontrar una elemento en la lista, saber si todos los elementos observan una propiedad...

Lo que veis es el principio básico del algoritmo MapReduce: transformo valores con `map` y reduzco el resultado con `fold` (que en otros lenguajes se suele denominar `reduce`). Cuando los datos que resultan del `map` son de un tipo que es instancia de `Monoid` (que no nos da tiempo a ver en esta charla), es posible efectuar las reducciones en procesadores distintos, en paralelo.

Vamos a refactorizar. Podemos efectuar un par de eta-reducciones (supresiones de parámetros y argumentos cuando ambos ocupan la última posición a los dos lados de la definición) y el `fmap` queda más compacto si recurrimos al operador aplicativo `<$>`.

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> Maybe [Frame]
toFrames = go 1
  where
    go 10 [x, y]
      | x + y < 10 = Just [Open x y]
      | otherwise  = Nothing
    go 10 [x, y, z]
      | x     == 10 = Just [Strike y z]
      | x + y == 10 = Just [Spare x z]
      | otherwise   = Nothing
    go n (x:y:z:ys)
      | x     == 10 = (Strike y z :) <$> go (n+1) (y:z:ys)
      | x + y == 10 = (Spare x z :) <$> go (n+1) (z:ys)
      | x + y < 10  = (Open x y  :) <$> go (n+1) (z:ys)
      | otherwise   = Nothing
    go _ _ = Nothing

frameScore :: Frame -> Int
frameScore (Open x y)   = x + y
frameScore (Spare _ y)  = 10 + y
frameScore (Strike x y) = 10 + x + y

score :: [Frame] -> Int
score = sum . map frameScore
```

Hemos escito `score` en formato *point free*, combinando funciones con el operador `(.)`. El operador de composición de funciones se define así:

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

Nos permite combinar funciones de un modo similar a como formamos tubería en el shell de Unix, solo que en el sentido de derecha a izquierda. Podemos calcular el logaritmo natural de la raíz cuadrada de un número así:

```
Prelude> log (sin 1.0)
-0.17260374626909167
```

o así:

```
Prelude> (log . sin) 1.0
-0.17260374626909167
```

Las funciones que se definen haciendo uso de la composición con `(.)` pueden omitir el parámetro. Es decir, las siguientes dos definiciones son equivalentes:

```
score = sum . map frameScore

score xs = sum (map frameScore xs)
```

La primera de las dos definiciones está en estilo *point free* (aunque parece contradictorio porque es la que lleva un punto para componer las funciones).

3.2.16 Combinación de la conversión a frames con el cálculo de puntuación

Podemos, finalmente, crear una función que combine `toFrames` y `score`:

```
module Bowling where

data Frame = Open Int Int
           | Spare Int Int
           | Strike Int Int
           deriving (Eq, Show)

toFrames :: [Int] -> Maybe [Frame]
toFrames = go 1
  where
    go 10 [x, y]
      | x + y < 10 = Just [Open x y]
      | otherwise  = Nothing
    go 10 [x, y, z]
      | x     == 10 = Just [Strike y z]
      | x + y == 10 = Just [Spare x z]
      | otherwise   = Nothing
    go n (x:y:z:ys)
      | x     == 10 = (Strike y z :) <$> go (n+1) (y:z:ys)
      | x + y == 10 = (Spare x z :) <$> go (n+1) (z:ys)
      | x + y < 10  = (Open x y  :) <$> go (n+1) (z:ys)
      | otherwise   = Nothing
    go _ _ = Nothing

frameScore :: Frame -> Int
frameScore (Open x y)   = x + y
frameScore (Spare _ y)  = 10 + y
frameScore (Strike x y) = 10 + x + y

score :: [Frame] -> Int
score = sum . map frameScore

scorePlay :: [Int] -> Maybe Int
scorePlay pins = do
  frames <- toFrames (return $ score frames)
  return $ score frames
```

La notación de `scorePlay` es especial: aparece un `do`, una flecha invertida (`<-`) que recuerda a la asignación y lo que estamos acostumbrado a interpretar como una palabra clave pero que en Haskell es una función (`return`). Es la notación `do` para tratamiento de mónadas, útil cuando combinamos funciones de la forma `a -> m b` y `m` es una mónada. En nuestro caso combinamos las funciones `toFrames :: [Int] -> Maybe Frame` y `score :: [Frame] -> Int`, que encajan en el patrón porque `Maybe` es una mónada. Pero nos estamos adelantando.

4 Functores, functores applicativos y mónadas

Incluso con programas tan sencillos como los de las katas, y sin forzar mucho la máquina, hemos tenido ocasión de introducir abstracciones funcionales. La familiaridad hace que detectar casos de uso de estas abstracciones sea más sencillo, naturalmente. Y el uso suele simplificar los programas. Para el neófito, el uso de `fmap` sobre tipos que pueden parecer esotéricos, como `Maybe`, puede ser fuente de confusión. Pero, pasado un tiempo, el código gana en legibilidad y se reduce la posibilidad de cometer errores. Aquí os pido un acto de fe. Es así, y punto.

Ahora vamos a profundizar algo en `fmap` y las estructuras que lo soportan: los functores. Pasaremos luego a estudiar un caso particular de functor que encuentra aplicaciones interesantes: el functor aplicativo. Y acabaremos con una introducción muy somera a las mónadas, un caso particular de functor aplicativo que conforma uno de los (meta) patrones de diseño más utilizados en Haskell.

4.1 Functor

Hay operaciones sobre listas que transforman su contenido pero mantienen la estructura (no alteran la longitud o posición relativa de los objetos). Algunos ejemplos:

```
amarzal$ ghci
GHCi, version 7.10.1: http://www.haskell.org/ghc/  :? for help
Prelude> let xs = [1..5]
Prelude> map (1+) xs
[2,3,4,5,6]
Prelude> map (\x -> x*x) xs
[1,4,9,16,25]
Prelude> map (<3) xs
[True,True,False,False,False]
Prelude> map (toLower) "aBcD!@9"
"abcd!@9"
```

`map` es una función con perfil `map :: (a -> b) -> [a] -> [b]` que, recordemos, podemos definir así: `map f xs = [f x | x <- xs]`.

Podemos ver la función `map`, esencialmente, como un bucle que transforma una lista de tipo `[a]` en otra de tipo `[b]` con una función de tipo `a -> b`. Pero la visión interesante es otra: `map` eleva una función de tipo `a -> b` del mundo "normal" al mundo de las "listas", es decir, la convierte en una función `[a] -> [b]`. Basta con usar paréntesis en el perfil de `map` para que esto sea evidente:

```
map :: (a -> b) -> ([a] -> [b])
```

Decimos que la lista es un functor por que permite establecer transformaciones entre tipos (paso de un tipo `a` al tipo `[a]` y del tipo `b` al tipo `[b]`) y entre funciones (pasa una función de tipo `a -> b` al tipo `[a] -> [b]`) observando unas determinadas propiedad (que presentamos más adelante).

Podemos abstraer la idea del functor con una clase:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

No vale cualquier estructura con una definición de `fmap` con ese perfil. Se deben observa, además unas condiciones.

- `fmap id = id`: si "mapeo" la función identidad sobre los elementos de un functor, no se efectúa cambio alguno.
- `fmap (p . q) = fmap p . fmap q`: da igual mapear la composición de funciones que componer el mapeo de cada función.

4.1.1 Listas

Las listas son functores definiendo `fmap` así (es decir, haciendo que `fmap` sea `map`):

```
instance Functor [] where
  fmap f xs = [f x | x <-xs]
```

Veamos que se observan las dos leyes:

Primera ley:

```
fmap id xs
== -- aplicación de definición de fmap
[id x | x <- xs]
== -- aplicación de función id
[x | x <- xs]
==
xs
```

Segunda ley:

```
fmap (p . q) xs
== -- aplicación de definición de fmap
[(p . q) x | x <- xs]
== -- aplicación de definición de composición
[ p (q x) | x <- xs]
== -- Deshacemos la aplicación de un fmap
fmap p [q x | x <- xs]
== -- Deshacemos la aplicación del otro fmap
fmap p (fmap q [x <- xs])
==
fmap p (fmap q xs)
== Deshacemos aplicación de operador de composición
(fmap p . fmap q) xs
```

4.1.2 Maybe

Vamos con un functor interesante por su uso para el tratamiento de errores: `Maybe`.

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

Es trivial comprobar que esta definición de `fmap` observa las dos leyes. En el fondo, `Maybe` puede verse como una lista con cero elementos (`Nothing`) o con un elemento (`Just x`). Si el mapeo tenía sentido con listas de tamaño arbitrario, también lo tiene con listas de tamaño restringido.

Imaginemos una función que proporciona un valor posiblemente erróneo y que modelamos esta situación con `Maybe`:

```
safeLog :: Double -> Maybe Double
safeLog x | x < 0.0   = Nothing
          | otherwise = Just (log x)
```

¿Cómo defino una función que, dado un `x`, calcule `1 + log x` con seguridad? Puedo hacerlo por un camino pesado:

```
safe1PlusLog :: Double -> Maybe Double
safe1PlusLog x | x < 0.0   = Nothing
               | otherwise = Just (1 + log x)
```

o aprovechar que `Maybe` es un functor

```
safe1PlusLog = fmap (1+) . safeLog
```

Hay más functores, pero lo vamos a dejar aquí.

4.2 Functor aplicativo

Si aplico una función `a -> b -> c -> d` a un dato de tipo `a` obtengo una función `b -> c -> d`. Si aplico la función resultante a un dato de tipo `b`, obtengo una función de tipo `c -> d`. Y si aplico la nueva función a un dato de tipo `c`, obtengo un valor de tipo `d`. Esta componibilidad de la aplicación de funciones es uno de los ejes maestros en la construcción de programas funcionales. En el fondo, los programas no son más que la composición de funciones para formar una expresión cuya evaluación ofrece un resultado. Pero vamos a ver algunos problemas de componibilidad con functores y como hay una estructura más potente que les da solución.

4.2.1 Listas

¿Cómo aplico una lista de funciones a una lista de argumentos?

```
aplicar [(1+), (2+), (3+)] [4,5,6]
```

Una definición posible sería aplicar `(1+)` a cada elemento de la lista, luego aplicar `(2+)` a cada elemento de la lista y, finalmente, aplicar `(3+)` a cada elemento de la lista[2].

```
aplicar :: [a -> b] -> [a] -> [b]
aplicar (f:fs) (x:xs) = [f x | f <- fs, x <- xs]
```

El resultado de la expresión sería `[5,6,7,6,7,8,7,8,9]`. El problema no es tanto cómo definir la operación sino el hecho de que no podemos expresar cómodamente la composición.

¿Y si en lugar de funciones unarias como `(1+)`, `(2+)` o `(3+)` a una lista de valores, quiero aplicar una función ternaria a tres listas de valores?

```
aplicar3 (\ x y z -> x + y * z) [1,2,3] [10,20,30] [100,200,300]
```

He de definir una nueva función `aplicar3`, ya que `aplicar` solo vale para funciones binarias. Sería mejor disponer de una caja de herramientas que facilite la aplicación de funciones de aridad n a n listas de argumentos.

Aquí es donde entran en juego los functores aplicativos.

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

Leyes:

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

La función `pure` mete a la función que se le pasa en el "contexto" mínimo necesario (en el caso de la lista, la lista; en el caso de `Maybe`, en un `Just`). El operador `<*>` es la aplicación de la(s) función(es) "contextualizadas" a un argumento.

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

Es decir, estoy definiendo qué hacer al aplicar una lista de funciones a una lista de argumentos. El resultado puede ser una lista de valores o una lista de nuevas funciones que aplicar a la siguiente lista de argumentos. Ahora sí hay una buena composición:

[2] No es la única posibilidad. También podríamos aplicar la primera función al primer dato, la segunda al segundo y así sucesivamente. En el ejemplo obtendríamos como resultado la lista `[5,7,9]`. Las listas pueden definirse como functores aplicativos de más de un modo, aunque solo estudiamos aquí uno de ellos.

```
pure (+) <*> [1, 2, 3] <*> [4, 5, 6]
==
[(+)] <*> [1, 2, 3] <*> [4, 5, 6]
==
[(1+), (2+), (3+)] <*> [4, 5, 6]
==
[5, 6, 7, 6, 7, 8, 7, 8, 9]
```

o lo que es lo mismo:

```
Prelude> [(+)] <*> [1,2,3] <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]
```

Un operador que ayuda al trabajo con functores aplicativos es `<$>`, que se define así:

```
f <$> xs = pure f <*> xs
```

aunque basta con definirlo como `(<$>) = fmap`.

Las expresiones pasan a ser más legibles (¡una vez te acostumbras!)

```
Prelude> (+) <$> [1,2,3] <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]
Prelude> (\ x y z -> x + y * z) <$> [1,2,3] <*> [10,20,30] <*> [100,200,300]
[1001,2001,3001,2001,4001,6001,3001,6001,9001,1002,2002,3002,2002,4002,6002,3002,6002,9002,1003,2003,3003,2003,4003,60
Prelude> [(\ x y z -> x + y * z)] <*> [1,2,3] <*> [10,20,30] <*> [100,200,300]
[1001,2001,3001,2001,4001,6001,3001,6001,9001,1002,2002,3002,2002,4002,6002,3002,6002,9002,1003,2003,3003,2003,4003,60
```

4.2.2 Maybe

No solo las listas son functores aplicativos. `Maybe` también es un functor aplicativo.

```
instance Applicative Maybe where
    pure f = Just f
    Just f <*> Just x = Just x
    _ <*> _           = Nothing

Just (1+) <*> Just 3              -- resultado: Just 4
(1+) <$> Just 3                   -- resultado: Just 4
Just (+) <*> Just 10 <*> Just 2   -- resultado: Just 12
(+) <$> Just 10 <*> Just 2        -- resultado: Just 12
(*) <$> safeLog 1 <*> safeLog 1   -- resultado: Just 0
(*) <$> safeLog 1 <*> safeLog (-1) -- resultado: Nothing
```

Así se simplifica enormemente el trabajo con funciones que podrían retornar valores erróneos.

4.2.3 Análisis sintáctico aplicativo

Si has llegado hasta aquí pensarás que esto de los functores aplicativos es un mero juego intelectual que no puede tener ninguna aplicación práctica. Espero que este apartado te convenza de que esa abstracción es realmente útil: vamos a diseñar analizadores sintácticos aplicativos.

¿Qué es analizador sintáctico? En su versión más simple, es una función que, dada una cadena, proporciona un valor de cierto tipo (por ejemplo, un árbol sintáctico o semántico) que es descrito por esa cadena.

```
parser :: String -> t
```

El analizador puede no consumir la cadena completa. Nos vendrá bien que consuma un prefijo, produzca el valor asociado al prefijo y nos indique el sufijo pendiente de análisis:

```
parser :: String -> (t, String)
```

Ocurre que es posible que el analizador se enfrente a una cadena defectuosa, en cuyo caso será incapaz de producir un dato válido. Este perfil parece más adecuado:

```
parser :: String -> Maybe (t, String)
```

Pero también puede que una misma cadena dé lugar a más de un resultado, es decir, una lista de resultados (por ambigüedades en el análisis, por ejemplo). Si asumimos que los errores proporcionan una lista vacía en lugar de `Nothing` y que los análisis exitosos son elementos de la lista, este otro perfil es más apropiado:

```
parser :: String -> [(t, String)]
```

Como vamos a usar analizadores como valores, definamos el tipo apropiado:

```
newtype Parser t =
  Parser { runParser :: String -> [(t, String)] }
```

La definición del tipo con `newtype` en lugar de con `data` solo tiene por objeto hacer más eficiente el programa. Con `data`, cada constructor se codifica en el programa en ejecución de modo explícito. Si un tipo `data` solo tiene un constructor, podemos usar `newtype` para asegurar que el compilador aplica la comprobaciones de tipos rigurosamente pero sin sobrecoste temporal o espacial en ejecución.

Hemos usado notación de registros con un selector de nombre un tanto curioso. Se llama `runParser` porque cuando queramos acceder a la función almacenada en el `Parser t` lo haremos para aplicarla a un valor. Si `p` es un dato de tipo `Parser t`, quedará bien usarlo así `runParser p "cadena"`. Hasta que te acostumbres a esta forma de nombrar campos en ciertas construcciones, lo pasarás mal.

4.2.3.1 Analizadores básicos

Vamos a construir el analizador más sencillo: uno que consume la letra que le indiquemos:

```
letra :: Char -> Parser Char
letra l = Parser (\input -> case input of
                              (c:cs) | c == l -> [(c, cs)]
                              otherwise       -> []
                 )
```

Usamos así el analizador:

```
Prelude> runParser (letra 'a') "abcdef"
('a', "abcdef")
```

Este otro analizador siempre tiene éxito y devuelve lo que nos diga el usuario, sin consumir ningún carácter de la entrada:

```
success :: a -> Parser a
success x = Parser (\input -> [(x, input)])
```

4.2.3.2 Los analizadores son functores

No cuesta mucho definir `fmap` sobre analizadores:

```
instance Functor Parser where
  fmap f (Parser p) = Parser (\input -> [(f x, cs) | (x, cs) <- p input])
```

Esto nos permite aplicar funciones al resultado del analizado. Si mapeamos la función `ord :: Char -> Int` sobre el primer analizador básico,

```
Prelude> runParser (fmap ord $ letra 'a') "abcdef"
(97, "abcdef")
Prelude> let ascii = fmap ord . letra
Prelude> runParser $ ascii 'a' "abcdef"
(97, "abcdef")
```

4.2.3.3 Los analizadores son functores aplicativos

¿Cómo concatenar dos analizadores? Es decir, ¿cómo uso un segundo analizador cuando acaba con éxito el primero y justo desde el punto en el que él dejó un sufijo pendiente de análisis? ¿Y si en lugar de concatenar dos analizadores quiero concatenar tres, o cuatro?

Pensemos en un primer perfil para una función que de momento llamaremos `concatena`:

```
concatena :: Parser a -> Parser b -> Parser (a, b)
concatena (Parser p1) (Parser p2) =
  Parser (\input -> [((x, y), cs') | (x, cs) <- p1 input, (y, cs') <- p2 cs)])
```

Esta definición es válida, pero da problema en términos de composición. Imaginemos un uso como este:

```
p1 `concatena` p2 `concatena` p3
```

donde `p1 :: Parser a`, `p2 :: Parser b` y `p3 :: Parser c`. El resultado es un `Parser ((a, b), c)`. Por ese camino vamos a llegar a resultados muy complejos, con un anidamiento extraordinario.

Por ejemplo:

```
let p = concatena (letra 'a') (letra 'b')
in  runParser p "abcdef"
```

Hemos creado un parser que detecta el prefijo `"ab"` en la cadena "abcdef" y nos devuelve como resultado `(('a', 'b'), "cdef")`. Preferiríamos que no devolviese un par como primer componente, sino una lista. ¿Cómo lo hacemos?

No deberíamos devolver un par como forma de composición de los resultados. Sería mejor que el usuario proporcionara una función indicando cómo combinar los resultados de los analizadores.

La función que combina los resultados estará definida, por ejemplo, como de tipo `a -> b -> c` y lo que buscamos es "elevarla" al mundo de los analizadores, es decir, convertirla en una función `Parser a -> Parser b -> Parser c`.

¿Cómo definiríamos `concatena2`?

```
concatena2 :: (a -> b -> c) -> Parser a -> Parser b -> Parser c
concatena2 f (Parser p1) (Parser p2) =
  Parser (\input -> [(f r1 r2, cs') | (r1, cs) <- p1 input, (r2, cs') <- p2 cs])
```

Ahora

```
let p = concatena2 (\(x,y) -> [x, y]) (letra 'a') (letra 'b')
in  runParser p "abcdef"
```

devuelve lo que esperamos: `("ab", "cdef")`

No está mal, pero surge un problema cuando tratamos de combinar más de dos analizadores. Imagina que queremos reconocer el prefijo "abc":

```
concatena3 (\ x y z -> [x, y, z]) (letra 'a') (letra 'b') ??? (letra 'c')
```

¿Cómo "llego" al tercer analizador? La función combina solo admite dos operandos. Una vía sería usar varios `concatena3`:

```
let p = concatena2 (\ x y -> [x, y]) (letra 'a') (letra 'b')
    p' = concatena2 (\x y -> x ++ [y]) p (letra 'c')
in runParser p' "abcdef"
```

Lo hemos conseguido, pero el resultado no es muy legible. Y si queremos ir más allá tendremos que definir combinadores especiales para dos, tres, cuatro analizadores... Poco elegante.

```
concatena3 :: (a -> b -> c -> d) -> Parser a -> Parser b -> Parser c -> Parser d
concatena3 f (Parser p1) (Parser p2) (Parser p3) =
    Parser (\input ->  [(f x y z, cs”) | (x, cs)   <- p1 input
                                       , (y, cs’)  <- p2 cs
                                       , (z, cs”) <- p3 cs’])

let p = concatena3 (\x y z -> [x,y,z]) (letra ’a’) (letra ’b’) (letra ’c’)
in  runParser p ”abcdef”
```

Este problema es, básicamente, el que teníamos al combinar listas de funciones con listas de argumentos. Y la solución es la misma: definir el analizador como functor aplicativo.

```
class Applicative Parser where
    pure :: a -> f a
    pure = success

    (<*>) :: f (a -> b) -> f a -> f b
    Parser f <*> Parser p = Parser (\input -> [(f x, cs) | (x, cs) <- p input])
```

Vamos a concatenar tres analizadores:

```
let p = pure (\x y z -> [x, y, z]) <*> (letra ’a’) <*> (letra ’b’) <*> (letra ’c’)
in runParser p ”abcdef”
```

El resultado es el par `(”abc”, ”def”)`

¿Por qué funciona? Si el primer analizador es de tipo `Parser (a -> b -> c)` y el segundo es de tipo `Parser a`, el resultado es un analizador de tipo `Parser (b -> c)`, así que lo podemos combinar con un analizador de tipo `Parser b` para producir un analizador de tipo `Parser c`.

4.2.4 El combinador de alternativas

Para analizar gramáticas incontextuales nos falta un operador más: el que permite optar entre dos alternativas:

```
(<|>) :: Parser a -> Parser a -> Parser a
Parser p1 <|> Parser p2 = Parser (\input -> p1 input ++ p2 input)
infixr 3 <|>
```

4.2.5 Un ejemplo de gramática y análisis

Vamos con un ejemplo de análisis algo más complicado: el reconocimiento de secuencias de paréntesis bien anidados con cálculo de la mayor profundidad de anidamiento. Una cadena como `”(()(()))”` ofrece como resultado el valor 2. La gramática incontextual que modela ese lenguaje es

$$S \rightarrow '(' \; S \; ')' \; S \mid \epsilon$$

Y esta su traducción con analizadores aplicativos:

```
parens :: Parser Int
parens = success (\ _ b _ d -> (1+b) `max` d)
         <*> letra ’(’ <*> parens <*> letra ’)’ <*> parens
         <|> success 0
```

Si aplicamos el analizador a la cadena `”(()(()))”` obtenemos este resultado:

```
[(2,””),(1,”(()(()))”),(0,”(()(()))”)]
```

El análisis completo ha calculado correctamente la profundidad.

Los aplicativos tienen muchas más aplicaciones, valga el juego de palabras. Lo importante es que veamos el tipo de abstracciones a que invita la programación funcional muy diferente de las que estas acostumbrado a encontrar en otros lenguajes. La programación funcional formula preguntas "nuevas" y las respuestas a esas preguntas responden familias de problemas que no parecen conectados entre sí.

Cuando te hayas metido a fondo en Haskell, te resultará muy útil la lectura de Combinator Parsing: A Short Tutorial, de S. Doaitse Swierstra. Este apartado se ha basado en ese texto.

4.3 Mónadas

Y vamos con las mónadas, una de las estructuras que acaban fascinando a quienes se aproximan a la programación funcional y el tema del encontraréis más tutoriales en la red (algunos bastante extravagantes).

4.3.1 La componibilidad de `Maybe`

Sabes que hay un combinador de funciones muy utilizado en Haskell:

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

Es posible expresar cálculos "en tubería" con ese operador. Por ejemplo, si tenemos una lista de enteros y queremos convertir su primer elemento en el carácter ese código ASCII

```
Prelude> import Data.Char
Prelude Data.Char> chr . head $ [48, 49, 50]
'0'
```

La función `head` tiene perfil `[a] -> a` y la función `chr` tiene perfil `Int -> Char`. Cuando `head` se aplica a una lista de enteros, el perfil se instancia a `[Int] -> Int`, así que podemos componer fácilmente ambas funciones.

Pero, ¿qué pasa cuando tratamos de aplicar este cálculo a una lista vacía?

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

Se ha producido una excepción, que es un efecto secundario, algo que hemos de evitar a toda costa. La culpa no la tiene el operador de composición, sino la función `head`, que no es "segura":

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

Podemos diseñar una función de acceso al primer elemento de una lista más segura:

```
safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead []    = Nothing
```

Pero ahora tenemos un problema de componibilidad, una impedancia en la conexión de las funciones, ya que `safeHead` aplica a un dato de tipo `[Int]` y devuelve un `Maybe Int`, que es un dato de un tipo "incompatible" con la entrada de `chr`, que es una función que es espera un simple `Int` como argumento. Pero es que `chr` también es problemática y tiene efectos secundarios:

```
Prelude Data.Char> chr (-1)
*** Exception: Prelude.chr: bad argument: (-1)
```

Debería diseñar una función segura para `chr`:

```haskell
safeChr :: Int -> Maybe Char
safeChr x
  | x >= 0    = Just $ chr x
  | otherwise = Nothing
```

Recordemos que `Maybe` es un functor, así que puedo usar `fmap` para combinar el `Maybe Int` que resulta de `safeHead` con la función `safeChr`:

```haskell
fmap safeChar $ safeHead [48, 49, 50]
```

Ejecutemos este programa con `runhaskell`:

```haskell
import Data.Char

safeChr :: Int -> Maybe Char
safeChr x
  | x >= 0    = Just $ chr x
  | otherwise = Nothing

safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead []    = Nothing

main :: IO ()
main = do
  print $ fmap safeChr $ safeHead [48, 49, 50]
  print $ fmap safeChr $ safeHead []
  print $ fmap safeChr $ safeHead [-1]
```

En pantalla veremos los siguiente:

```
Just (Just '0')
Nothing
Just Nothing
```

El resultado no es un `Maybe Char`, sino un `Maybe (Maybe Char)`. Deberíamos de poder simplificar `Just (Just '0')` como `Just '0'` y `Just Nothing` como `Nothing`.

Definamos una operación `join` que funda un `Maybe (Maybe a)` en un `Maybe a`:

```haskell
import Data.Char

safeChr :: Int -> Maybe Char
safeChr x
  | x >= 0    = Just $ chr x
  | otherwise = Nothing

safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead []    = Nothing

join :: Maybe (Maybe a) -> Maybe a
join (Just x)  = x
join Nothing = Nothing

main :: IO ()
main = do
  print $ join $ fmap safeChr $ safeHead [48, 49, 50]
  print $ join $ fmap safeChr $ safeHead []
  print $ join $ fmap safeChr $ safeHead [-1]
```

La ejecución proporciona este resultado:

```
Just '0'
Nothing
Nothing
```

Una mónada puede verse como una estructura que permite automatizar la operación de `fmap` con `join`, pero suele presentarse de otro modo.

```
class Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    return  :: a -> m a
```

El operador `mx >>= f`, al que se denomina *bind*, no es más que `(join . fmap f) mx`.

¿Qué interés tiene? En nuestro caso, simplifica el encadenamiento de funciones que pueden fallar, por ejemplo.

Las mónadas son tan importantes que tienen su propia notación en Haskell. Podemos encadenar cálculo de un modo que parece imperativo:

```
import Data.Char

safeChr :: Int -> Maybe Char
safeChr x
  | x >= 0    = Just $ chr x
  | otherwise = Nothing

safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead []    = Nothing

chrAtHead :: [Int] -> Maybe Char
chrAtHead list = do
  x <- safeHead list
  y <- safeChr x
  return y

main :: IO ()
main = do
  print $ chrAtHead [48, 49, 50]
  print $ chrAtHead []
  print $ chrAtHead [-1]
```

La notación do hilvana las líneas con llamadas a >>= para evitar escribir esto otro:

```
import Data.Char

safeChr :: Int -> Maybe Char
safeChr x
  | x >= 0    = Just $ chr x
  | otherwise = Nothing

safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead []    = Nothing

chrAtHead :: [Int] -> Maybe Char
chrAtHead list = safeHead list >>= safeChr

main :: IO ()
```

```haskell
main = do
  print $ chrAtHead [48, 49, 50]
  print $ chrAtHead []
  print $ chrAtHead [-1]
```

Como dicen algunos: Haskell es el mejor lenguaje imperativo del mundo.

4.3.2 List

Las listas también son mónadas. Un enfoque que permite visualizar las listas como mónadas es el modelado del no determinismo. Imaginemos una función cuyo valor de en los de un conjunto de posibilidades. El conjunto puede representarse con una lista.. Para ilustrarlo, imaginemos una función que podría incrementar o decrementar el valor de un entero:

```haskell
incOrDec :: Int -> [Int]
incOrDec x = [x-1, x+1]
```

Volvemos a tener problemas de composición si queremos aplicar la propia función al resultado de aplicar la propia función. Una aplicación naíf de `incOrDec` a una lista de enteros no producirá una lista, sino una lista de listas:

```haskell
naif :: [Int] -> (Int -> [Int]) -> [[Int]]
naif lista f = [f x | x <- lista]
```

```
Prelude> let incOrDec x = [x-1, x+1]
Prelude> let r = incOrDec 5
Prelude> r
[4,6]
Prelude> let naif lista f = [f x | x <- lista]
Prelude> naif r incOrDec
[[3,5],[5,7]]
```

En realidad lo que queremos es una lista de valores, no una lista de listas. Las mónadas vienen en nuestro auxilio:

```
Prelude> import Control.Monad
Prelude Control.Monad> incOrDec 5 >>= incOrDec
[3,5,5,7]
```

4.3.3 Entrada/salida

La entrada salida es problemática porque rompe la transparencia referencial. La lectura de una línea por teclado se hace con la función `getLine`. Supongamos que el perfil de `getLine` fuese `() -> String` (que no lo es).

¡Pero cada vez que llamamos a `getLine` obtenemos un resultado distinto!

Es más, si el orden de evaluación del código no está predeterminado, si el usuario escribe en teclado primero “hola” y luego “mundo”, ¿cómo sé que saldrá resultará de evaluar esta expresión?

```haskell
getLine () ++ getLine ()
```

¿Resultará “holamundo” o “mundohola”?

4.3.3.1 El mundo como parámetro

No habría problema si `getLine` recibiera el mundo real como un argumento:

```
getLine :: World -> String
```

La definición tiene un problema: leer una línea de teclado cambia el estado del mundo real. Debería tener acceso al nuevo estado del mundo real:

```
getLine :: World -> (String, World)
```

Demos un nombre al tipo `World -> (a, World)` por concisión:

```
newtype IO a = IO (World -> (a, World))
```

Podemos definir el perfil de `getLine` así:

```
getLine :: IO String
```

Cuando quiera hacer dos llamadas a `getLine` y el estado actual del mundo esté almacenado en una variable `world`, actuaré así:

```
let (linea1, world')  = getLine world
    (linea2, world") = getLine world'
in linea1 ++ linea2
```

Aquí sí que hay un orden de evaluación preciso: el que ordenan las dependencias entre valores.

El problema es lo farragoso de tener que llevar manualmente el control sobre el estado del mundo. La cosa se simplifica si defino un modo de composición de funciones `IO String`. La aproximación monádica es de ayuda:

```
instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  (IO io) >>= f = IO (\world -> let (r, world') = io world
                                in f r world')
  return x = \world -> (x, world)
```

Gracias a esta visión monádica podemos escribir código en el que el orden de las líneas indique orden de evaluación:

```
do linea1 <- getLine
   linea2 <- getLine
   return (linea1 ++ linea2)
```

4.3.3.2 El programa principal

El programa principal tiene este perfil:

```
main :: IO ()
```

Es decir, el programa recibe el estado del mundo como argumento y devuelve el estado del mundo tal cual queda tras la aplicación de la función. Curioso. Y otra cosa curiosa: los programas que tienen efectos secundarios devuelven la mónada `IO`. Una vez entras en la mónada `IO`, no puedes salir de ella. El código separa con claridad las zonas seguras de las inseguras.

5 Fin

5.1 Qué no cabía y me hubiera gustado contar

Algunas estructuras básicas, como `Monoid`, para MapReduce. Extenderme más en map-filter-reduce. Ver más mónadas, en particular `Reader` (inyección de dependencias), `Writer` (logging con monoides) y `State` (del quie `IO` es un caso particular y con el que también se puede ilustrar la generación de números aleatorios). Las mónadas de concurrencia/paralelismo. Los transformadores de mónadas. Las estructuras de datos persistentes que vienen de serie y algunas otras. Algunas "perlas" de la programación funcional. Montar algo con la librería `Diagrams` para ilustrar gráficamente parte de lo expuesto. La metaprogramación con *quasiquotes*. Hablar de Lenses. La diferenciación automática (o algorítmica) como ilustración de lo elegante y conciso de Haskell. Las extensiones más comunes del lenguaje. El tratamiento eficiente de texto. El manejo de `ghci`. Los *kinds*. Las librerías de uso común. La exportación selectiva de funciones, nombres y tipos (y su ocultación). Lo que hay en el preludio (que ha cambiado en la versión 7.10 de ghc).

5.2 Qué hacer ahora

Lecturas recomendadas:

- Learn you a Haskell for great good, de Miran Lipovača.
- Beginning Haskell, de Alejandro Serrano Mena
- Thinking Functionally with Haskell, de Richard Bird
- Real World Haskell, de Richard Bird.
- Parallel and Concurrent Programming in Haskell, de Simon Marlow
- Write Yourself a Scheme in 48 Hours - A Haskell Tutorial, de Jonathan Tang.

Sigue, además, el agregador de blogs planet.haskell.org.

Haz el curso Introduction to Functional Programming de Erik Meijer, en EdX. O leete el material de este curso y practica con sus ejercicios: CIS 194: Introduction to Haskell (Spring 2013), de Brent Yorgey. O haz este otro: Functional Programming Course - NICTA, de Tony Morris. O pásate por el School of Haskell, de FPComplete. Repasa estas notas.

Pero, sobre todo, escribe código. Si no escribes código y solo lees, es imposible aprender a un ritmo razonable, *muy especialmente con Haskell*. Instálate Haskell. Bájate los paquetes de modo-Haskell pare tu editor favorito. Arranca con un proyecto personal o una pequeña kata. Empieza ya mismo.