2024’A

PROGRAMACIÓN LÓGICA Y FUNCIONAL

INSTITUTO TECNOLOGICO DE OAXACA

Conceptos Fundamentales.

Introducción

Sobre esta guía

Bienvenido a aprende programación lógica y funcional .Si estás leyendo esto probablemente quieras aprender a programar lógica y funcionalmente . Pues bien, pero primero vamos a hablar un poco sobre esta guía.

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

Aprender un nuevo lenguaje de programación siempre es extraño, confuso y algo estresante pero después es cómodo a paseo a lo que te gusta hacer y practicar, aprender algo nuevo es genial y si estás interesado en la programación funcional deberías aprenderlo incluso aunque te sea totalmente extraño. Aprender un lenguaje funcional es como aprender a programar por primera vez ¡Es divertido! Te fuerza a que pienses diferente, lo cual nos lleva a la siguiente sección...

Entonces, ¿qué es programación?

Programación: la programación es una colaboración entre humanos y computadoras

Programador: El programador escribe líneas de texto o "código" que se distribuyen en diferentes archivos dentro de una carpeta, siguiendo las reglas que le define un lenguaje de programación y que, finalmente, se ejecutan para cumplir una o múltiples funciones.

Lenguajes de programación: Existen diversos lenguajes de programación: cada uno define un conjunto de reglas y cada uno tiene un propósito en específico, así que, dependiendo del proyecto o tarea que el programador deba ejecutar, se utilizará un lenguaje en específico.

“Un programador sigue las reglas de un lenguaje para comunicarse con la computadora y así definir las instrucciones a seguir para construir un programa o software.”

Habilidades que desarrolla un buen programador

Algunas de las aptitudes o habilidades que se desarrollan durante esta actividad son:

- Pensamiento lógico.
- Aptitud matemática.
- Atención al detalle.
- Desarrollo de nuevas tecnologías.
- Aprendizaje de diferentes lenguajes de programación.
- Análisis de problemas.
- Gestión del tiempo.

Como ves, la programación es una disciplina que cada vez cobra más fuerza y popularidad ya que está estrictamente relacionada al desarrollo de nuevas tecnologías y, se podría decir, que es la base de muchas de las profesiones y modelos económicos del futuro.

¿Cómo resolvemos un problema con programación?

Para resolver un problema necesitamos…

Paradigma: Un paradigma de programación es la clasificación, el estilo o la forma de programar. Es un enfoque para resolver problemas mediante el uso de lenguajes de programación. Dependiendo del lenguaje, la dificultad de usar un paradigma cambia.

Paradigmas de programación.

programación imperativa y programación declarativa

- ***imperativo***: en el que el programador instruye a la máquina cómo cambiar su estado,
  - ***procedimental:*** que agrupa las instrucciones en procedimientos,
  - **orientado a objetos**: que agrupa las instrucciones con la parte del estado en el que operan,

- ***declarativo*** en el que el programador simplemente declara las propiedades del resultado deseado, pero no cómo calcularlo

  - ***funcional*** en el que el resultado deseado se declara el valor de una serie de aplicaciones de función,
  - ***lógico*** en la que el resultado deseado se declara la respuesta a una pregunta sobre un sistema de hechos y reglas,
  - ***matemático*** en el que el resultado deseado se declara la solución de un problema de optimización
  - ***reactivo*** en el que se declara el resultado deseado con flujos de datos y la propagación del cambio

Programación imperativa

Es un paradigma de programación, está enfocada en cómo se ejecuta el código, define el control de flujo de manera explícita y cambia el estado de una máquina.

Definición de variables

Todo ***let, const o var*** es imperativo, cada vez que defines una constante estas definiendo control de flujo y, además, estás mutando el estado de la máquina, este puede ser después recolectado por el garbage collector, sin embargo sigue siendo una mutación que se encuentra en la memoria de la máquina y además define una variable que será utilizada más adelante para definir el flujo.

Decisiones

Todo if, else, else if, switch o instrucción que te permita a ti tomar una decisión en base a algo que ya definiste es considerado imperativo.

GARBAGE COLLECTOR

Es el proceso automatizado de eliminar Código que ya no se necesita ni se utiliza

Loops (bucles o siclos)

Todo loop controlado, ya sea **for**, **while**, **do while**, entre otros, donde en base a una condición debamos detener este loop, es considerado imperativo, incluso los **foreach** donde de manera explícita estemos indicando que vamos a iterar un arreglo. En definitiva, cualquier loop controlado, infinito, con condición de salida, sin condición de salida, break o continue son considerados imperativos.

Manejo de excepciones con try catch

Cuando definimos de manera explícita un try catch, estamos definiendo de manera imperativa por qué lado de nuestro código vaya la ejecución de este.

Programación declarativa

se determina automáticamente la vía de solución. Esto funciona siempre y cuando las especificaciones del estado final se definan claramente y exista un procedimiento de ejecución adecuado. Si se dan las dos condiciones, la programación declarativa es muy eficiente.

Como la programación declarativa no determina el "cómo", sino que funciona a un nivel de abstracción muy alto, este paradigma deja margen para la optimización. Si se ha desarrollado un procedimiento de ejecución mejor, el algoritmo integrado lo encuentra y lo aplica. En este sentido, el paradigma está muy preparado para el futuro porque, al escribir el código, no es necesario determinar el procedimiento según el cual se alcanza el resultado.

Los lenguajes de programación declarativa más conocidos son:

- Prolog
- Lisp
- Haskell
- Miranda
- Erlang
- SQL (en un sentido amplio)

Los distintos lenguajes declarativos se pueden subdividir, a su vez, en dos paradigmas, el de la programación funcional y el de la programación lógica.

Sin embargo, es habitual que en la práctica los límites se difuminen y que, a la hora de solucionar problemas, se apliquen tanto elementos de la programación imperativa, con sus subtipos de programación procedimental, modular y estructurada, como de la programación declarativa.

"Como resolver el problema"

Programación Imperativa

programación declarativa

Saca tu cafetera
Cargue la cafetera con el café

Agrega agua a la cafetera

Coloque la cafetera en la estufa
Encienda la estufa
Vierta en café en una taza
¿Te gusta el azúcar?

- Si es así, agregue azúcar
- Si no, no agregue azúcar

Beber café

Ir a la cafetería local y decirle al barista

"Me gustaría una taza de café con una cuchara de azúcar, por favor"

El barista prepara tu café, te da la taza y la bebes sin preocuparte por los detalles de cómo se hizo,

Tipos de datos

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

Int: Representa enteros, por lo que 7 puede ser un Int pero 7.2 no puede. Int está acotado, lo que significa que tiene un valor máximo y un valor mínimo. Normalmente en máquinas de 32bits el valor máximo de Int es 2147483647 y el mínimo -2147483648.

Integer Representa enteros también. La diferencia es que no están acotados así que pueden representar números muy grandes. Sin embargo, Int es más eficiente.

Float: es un numero real en coma flotante de simple precisión

Double: es un numero real en coma flotante de ¡ Doble precisión !

Bool: Es el tipo de booleano. Solo puede tener dos valores True o false.

Char: Representa un carácter, se define rodeado por comillas simples, una lista de caracteres es una cadena .

Ejercicios

Realiza a mano estos ejercicios en java

1) Declara un String que contenga tu nombre, después muestra un mensaje de bienvenida por consola. Por ejemplo: si introduzco «AlumnITO», me aparezca «Bienvenido AlumnITO».

2) Declara dos variables numéricas (con el valor que desees), muestra por consola la suma, resta, multiplicación, división y residuo (resto de la división).

3) Declara 2 variables numéricas (con el valor que desees), he indica cual es mayor de los dos. Si son iguales indicarlo también. Ves cambiando los valores para comprobar que funciona.

4) Dada una lista de nombres y edades de personas (“Joaquina-18 ”, “José-12”, “Alberto-29”, “Camila-2”), crea un informe que muestre el nombre y la edad de cada persona en una lista numerada.

5) Se te pide que escribas un programa que genere un informe de ventas para una tienda. Tienes los siguientes datos: una lista de productos vendidos

(nombre del producto, cantidad vendida y precio unitario) y el nombre del vendedor.

Datos para crear tu programa.

- Productos: Camisa, pantalón, zapatos
- Cantidad vendida: 10, 5 , 8
- Precio unitario: $25.5, $35.75,$178.75
- Nombre vendedor: Juan López

6) Dada una lista de números crea el método “obtenerCabeza() “ para obtener el primer elemento de la lista. Luego imprime este resultado.
    - {100,45,2,8,9,25,68,3,47,22,47}

7) Calcula el índice de masa corporal, que indique que si el IMC es menor que 18.5 entonces tienes infra peso, si estas entre 18.5 y 25 eres normal y si tienes 25 y 30 tienes sobrepeso y si tienes más de 30 eres obeso.

Modelo de Programación Funcional.

Introducción al modelo de programación funcional.

Aprende Haskell

¿qué es Haskell?

Haskell es un lenguaje de programación puramente funcional. En los lenguajes imperativos obtenemos resultados dándole la computadora una secuencia de tareas que luego éste ejecutará. Mientras las ejecuta, puede cambiar de estado. Por ejemplo, establecemos la variable a a 5, realizamos algunas tareas y luego cambiamos el valor de la variable anterior. Estos lenguajes poseen estructuras de control de flujo para realizar ciertas acciones varias veces (for, while...). Con la programación puramente funcional no decimos al computador lo que tiene que hacer, sino más bien, decimos como son las cosas.

El factorial de un número es el producto de todos los números desde el 1 hasta ese número, la suma de una lista de números es el primer número más la suma del resto de la lista, etc. Expresamos la forma de las funciones. Además no podemos establecer una variable a algo y luego establecerla a otra cosa. Si decimos que a es 5, luego no podemos decir que es otra cosa porque acabamos de decir que es 5 ¿Acaso somos unos mentirosos? De este modo, en los lenguajes puramente funcionales, una función no tiene efectos secundarios. Lo único que puede hacer una función es calcular y devolver algo como resultado. Al principio esto puede parecer una limitación pero en realidad tiene algunas buenas consecuencias: si una función es llamada dos veces con los mismos parámetros, obtendremos siempre el mismo resultado. A esto lo llamamos transparencia referencial y no solo permite al compilador razonar acerca de el comportamiento de un programa, sino que también nos permite deducir fácilmente (e incluso demostrar) que una función es correcta y así poder construir funciones más complejas uniendo funciones simples.

Perezoso

Haskell es perezoso. Es decir, a menos que le indiquemos lo contrario, Haskell no ejecutará funciones ni calculará resultados hasta que se vea realmente forzado a hacerlo. Esto funciona muy bien junto con la transparencia referencial y permite que veamos los programas como una serie de transformaciones de datos. Incluso nos permite hacer cosas interesantes como estructuras de datos infinitas. Digamos que tenemos una lista de números inmutables xs = [1,2,3,4,5,6,7,8] y una función doubleMe que multiplica cada elemento por 2 y devuelve una nueva

lista. Si quisiéramos multiplicar nuestra lista por 8 en un lenguaje imperativo he hiciéramos doubleMe(doubleMe(doubleMe(xs))), probablemente el computador recorrería la lista, haría una copia y devolvería el valor. Luego, recorrería otras dos veces más la lista y devolvería el valor final. En un lenguaje perezoso, llamar a doubleMe con una lista sin forzar que muestre el valor acaba con un programa diciéndote "Claro claro, ¡luego lo hago!". Pero cuando quieres ver el resultado, el primer doubleMe dice al segundo que quiere el resultado, ¡ahora! El segundo dice al tercero eso mismo y éste a regañadientes devuelve un 1 duplicado, lo cual es un 2. El segundo lo recibe y devuelve un 4 al primero. El primero ve el resultado y dice que el primer elemento de la lista es un 8. De este modo, el computador solo hace un recorrido a través de la lista y solo cuando lo necesitamos. Cuando queremos calcular algo a partir de unos datos iniciales en un lenguaje perezoso, solo tenemos que tomar estos datos e ir transformándolos y moldeándolos hasta que se parezcan al resultado que deseamos.

Bote

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

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

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

Qué necesitas para comenzar

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

Cargando...

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

Primeros pasos en haskell

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

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

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

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

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

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

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

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

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

La comprobación de igualdad se hace así:

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

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

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

GHCi nos está diciendo es que **"texto"** no es un número y por lo tanto no sabe como sumarlo a 5. Incluso si en lugar de **"texto"** fuera **"cuatro"**, **"four"**, o **"4"**, Haskell no lo consideraría como un número. **+** espera que su parte izquierda y derecha sean números. Si intentamos realizar **True == 5**,

GHCi nos diría que los tipos no coinciden. Mientras que **+** funciona solo con cosas que son consideradas números, **==** funciona con cualquiera cosa que pueda ser comparada. El truco está en que ambas deben ser comparables entre si. No podemos comparar la velocidad con el tocino. Daremos un vistazo más detallado sobre los tipos más adelante. Nota: podemos

hacer **5 + 4.0** porque **5** no posee un tipo concreto y puede actuar como un entero o como un número en coma flotante. **4.0** no puede actuar como un entero, así que **5** es el único que se puede adaptar.

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

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

```
Ejemplo:
ghci> succ 8
9
```

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

```
Ejemplo:

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

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

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

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

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

La gente que ya conoce algún lenguaje imperativo tiende a aferrarse a la idea de que los paréntesis indican una aplicación de funciones. Por ejemplo, en C, usas los paréntesis para llamar a las funciones como **foo()**, **bar(1)**, o **baz(3, "jaja")**. Como hemos dicho, los espacios son usados

para la aplicación de funciones en Haskell. Así que estas funciones en Haskell serían **foo**, **bar 1** y **baz 3 "jaja"**. Si ves algo como **bar (bar 3)** no significa que **bar** es llamado con **bar** y **3** como parámetros. Significa que primero llamamos a la función **bar** con **3** como parámetro para obtener un número y luego volver a llamar **bar** otra vez con ese número. En C, esto sería algo como **bar(bar(3))**.

Las primeras pequeñas funciones

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

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

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

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

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

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

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

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

Una introducción a las listas

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

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

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

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

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

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

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

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

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

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

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

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

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

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

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

Nota

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

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

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

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

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

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

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

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

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

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

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

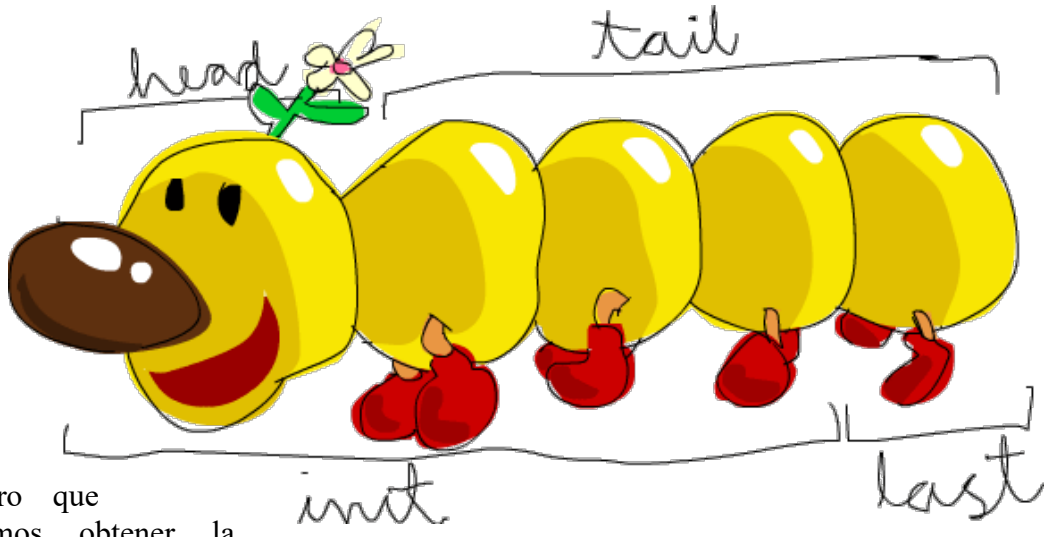
- last toma una lista y devuelve su último elemento.

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

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

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

Si imaginamos las listas como monstruos, serian algo como:



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

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

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

- length toma una lista y obviamente devuelve su tamaño.

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

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

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

- reverse pone del revés una lista.

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

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

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

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

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

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

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

- minimum devuelve el más pequeño.

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

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

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

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

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

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

Listas intencionales

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

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

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

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

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

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

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

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

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

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

```
ghci> boomBangs [7..13]

["BOOM!","BOOM!","BANG!","BANG!"]
```

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

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

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

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

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

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

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

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

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

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

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

Unas pruebas rápidas:

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

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

```
ghci> let xxs =
[[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]

ghci> [ [ x | x <- xs, even x ] | xs <- xxs]

[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

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

Tuplas

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

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

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

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

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

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

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

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

- fst toma una dupla y devuelve su primer componente.

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

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

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

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

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

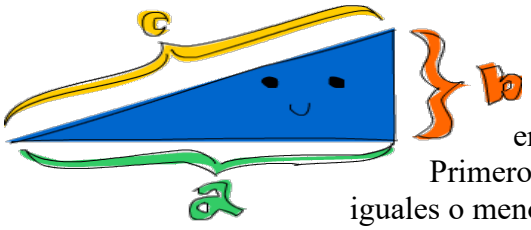
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5] ["uno","dos","tres","cuatro","cinco"]
[(1,"uno"),(2,"dos"),(3,"tres"),(4,"cuatro"),(5,"cinco")]
```

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

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

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

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



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


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

Simplemente estamos extrayendo valores de estas tres listas y nuestra función de salida las esta combinando en una tripla. Si evaluamos esto escribiendo **triangles** en GHCi, obtendremos una lista con todos los posibles triángulos cuyos lados son menores o iguales que 10. Ahora, debemos añadir una condición que nos filtre únicamente los triángulos rectos. Vamos a modificar esta

función teniendo en consideración que el lado b no es mas largo que la hipotenusa y que el lado a no es más largo que el lado b.

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

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

```
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2 == c^2, a+b+c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

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

Tipos y clases de tipos

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

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

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

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

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

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

Las funciones también tiene tipos. Cuando escribimos nuestras propias funciones podemos darles un tipo explícito en su declaración. Generalmente está bien considerado escribir los tipos explícitamente en la declaración de un función, excepto cuando éstas son muy cortas. De aquí en adelante les daremos tipos explícitos a todas las funciones que creemos. ¿Recuerdas la lista intensional que filtraba solo las mayúsculas de una cadena? Aquí tienes como se vería con su declaración de tipo:

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

removeNonUppercase tiene el tipo **[Char] -> [Char]**, que significa que es una función que toma una cadena y devuelve otra cadena. El tipo **[Char]** es sinónimo de **String** así que sería más

elegante escribir el tipo como **removeNonUppercase :: String -> String**. Anteriormente no le dimos un tipo a esta función ya que el compilador puede inferirlo por si solo. Pero, ¿cómo escribimos el tipo de una función que toma varios parámetros? Aquí tienes una función que toma tres enteros y los suma:

```
addThree :: Int -> Int -> Int -> Int

addThree x y z = x + y + z
```

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

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

Aquí tienes una descripción de los tipos más comunes:

- **Int** representa enteros. Se utiliza para representar número enteros, por lo que **7** puede ser un **Int** pero **7.2** no puede. **Int** está acotado, lo que significa que tiene un valor máximo y un valor mínimo. Normalmente en máquinas de 32bits el valor máximo de **Int** es 2147483647 y el mínimo -2147483648.
-
- **Integer** representa... esto... enteros también. La diferencia es que no están acotados así que pueden representar números muy grandes. Sin embargo, **Int** es más eficiente.

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

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

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

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

```
ghci> circumference 4.0
25.132742
```

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

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

```
ghci> circumference' 4.0
25.132741228718345
```

- **Bool** es el tipo booleano. Solo puede tener dos valores: **True** o **False**.
- **Char** representa un carácter. Se define rodeado por comillas simples. Una lista de caracteres es una cadena.

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

Variables de tipo

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

-> a

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

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

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

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

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

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

Clases de tipos paso a paso

Las clases de tipos son una especie de interfaz que define algún tipo de comportamiento. Si un tipo es miembro de una clase de tipos, significa que ese tipo soporta e implementa el comportamiento que define la clase de tipos. La gente que viene de lenguajes orientados a objetos es propensa a confundir las clases de tipos porque piensan que son como las clases en los lenguajes orientados a objetos. Bien, pues no lo son. Una aproximación más adecuada sería pensar que son como las interfaces de Java, o los protocolos de Objective-C, pero mejor.

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

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

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

Interesante. Aquí vemos algo nuevo, el símbolo =>. Cualquier cosa antes del símbolo => es una restricción de clase. Podemos leer la declaración de tipo anterior como: la función de igualdad toma dos parámetros que son del mismo tipo y devuelve un **Bool**. El tipo de estos dos parámetros debe ser miembro de la clase **Eq** (esto es la restricción de clase).
La clase de tipos **Eq** proporciona una interfaz para las comparaciones de igualdad. Cualquier tipo que tenga sentido comparar dos valores de ese tipo por igualdad debe ser miembro de la clase **Eq**. Todos los tipos estándar de Haskell excepto el tipo IO (un tipo para manejar la entrada/salida) y las funciones forman parte de la clase **Eq**.
La función **elem** tiene el tipo **(Eq a) => a -> [a] -> Bool** porque usa == sobre los elementos de la lista para saber si existe el elemento indicado dentro de la lista.
Algunas clases de tipos básicas son:

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

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

Ord es para tipos que poseen algún orden.

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

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

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

Para ser miembro de **Ord**, primero un tipo debe ser socio del prestigioso y exclusivo club **Eq**.
Los miembros de **Show** pueden ser representados por cadenas. Todos los tipos que hemos visto excepto las funciones forman parte de **Show**. la función más utilizada que trabaja con esta clase de tipos es la función **show**. Toma un valor de un tipo que pertenezca a la clase **Show** y lo representa como una cadena de texto

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

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

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

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

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

```
ghci> :t read
read :: (Read
```

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

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

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

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

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

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

minBound y **maxBound** son interesantes ya que tienen el tipo **(Bounded a) => a**. Es decir, son constantes polimórficas.
Todas las tuplas son también **Bounded** si sus componentes los son también.

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

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

```
ghci> :t 20

20 :: (Num t)
```

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

```
ghci> :t (*)
(*) :: (Num)
```

Toma dos números del mismo tipo y devuelve un número del mismo tipo. Esa es la razón por la que **(5 :: Int) * (6 :: Integer)** lanzará un error mientras que **5 * (6 :: Integer)** funcionará correctamente y producirá un **Interger**, ya que **5** puede actuar como un **Integer** o un **Int**.
Para unirse a **Num**, un tipo debe ser amigo de **Show** y **Eq**.
Integral es también un clase de tipos numérica. **Num** incluye todos los números, incluyendo números reales y enteros. **Integral** únicamente incluye números enteros. **Int** e **Integer** son miembros de esta clase.
Floating incluye únicamente números en coma flotante, es decir **Float** y **Double**.
Una función muy útil para trabajar con números es **fromIntegral**. Tiene el tipo **fromIntegral :: (Num b, Integral a) => a -> b**. A partir de esta declaración podemos decir que toma un número entero y lo convierte en un número más general. Esto es útil cuando estas trabajando con números reales y enteros al mismo tiempo. Por ejemplo, la función **length** tiene el tipo **length :: [a] -> Int** en vez de tener un tipo más general como **(Num b) => length :: [a] -> b**. Creo que es por razones históricas o algo parecido, en mi opinión, es absurdo. De cualquier modo, si queremos obtener el tamaño de una lista y sumarle **3.2**, obtendremos un error al intentar sumar un entero con uno en coma flotante. Para solucionar esto, hacemos **fromIntegral (length [1,2,3,4]) + 3.2**.

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

La sintaxis de las funciones

Ajuste de patrones

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

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

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

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

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

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

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

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

Esta es la primera vez que definimos una función recursiva. La recursividad es muy importante en Haskell, pero hablaremos de ello más adelante. Resumiendo, esto es lo que pasa cuando intentamos obtener el factorial de, digamos 3.

Primero intenta calcular **3 * factorial 2**. El factorial de 2 es **2 * factorial 1**, así que ahora tenemos **3 * (2 * factorial 1)**. **factorial 1** es **1 * factorial 0**, lo que nos lleva a **3 * (2 * (1 * factorial 0))**. Ahora viene el truco, hemos definido el factorial de 0 para que sea simplemente 1, y como se encuentra con ese patrón antes que el otro más general obtenemos 1. Así que el resultado equivale a **3 * (2 * (1 * 1))**. Si hubiésemos escrito el segundo patrón al inicio, hubiese aceptado todos los números incluyendo el 0 y el cálculo nunca terminaría. Por este motivo el orden es importante a la hora de definir los patrones y siempre es mejor definir los patrones más específicos al principio dejando los más generales al final.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Comprobamos que funciona:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Guardas

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

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

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

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

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

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

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

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

Vamos a ver si estoy gordo...

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

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

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

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

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

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

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

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

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise = LT
ghci> 3 `myCompare` 2
GT
```

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

¿Dónde?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Let it be

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Recursión

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

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

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

El impresionante maximum

La función **maximum** toma una lista de cosas que pueden ser ordenadas (es decir instancias de la clase de tipos **Ord**) y devuelve la más grande. Piensa en como implementaríamos esto de

forma imperativa. Probablemente crearíamos una variable para mantener el valor máximo hasta el momento y luego recorreríamos los elementos de la lista de forma que si un elemento es mayor que el valor máximo actual, lo remplazaríamos. El máximo valor que se mantenga al final es el resultado. ¡Wau! son muchas palabras para definir un algoritmo tan simple.

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

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

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

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

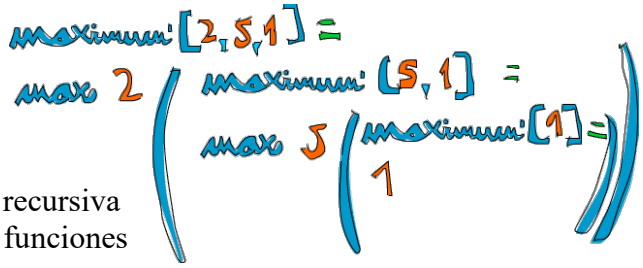
Vamos a tomar una lista de números de ejemplo y comprobar como funcionaria: **[2,5,1]**. Si llamamos **maximum'** con esta lista, los primeros dos patrones no ajustarían. El tercero si lo haría y la lista se dividiría en **2** y **[5,1]**. La sección **where** requiere saber el máximo de **[5,1]** así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

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

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

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

Unas cuantas más funciones recursivas



Ahora que sabemos cómo pensar de forma recursiva en general, vamos a implementar unas cuantas funciones de forma recursiva. En primer lugar, vamos a implementar **replicate**. **replicate** toma un **Int** y algún elemento y devuelve una lista que contiene varias repeticiones de ese mismo elemento. Por ejemplo, **replicate 3 5** devuelve **[5,5,5]**. Vamos a pensar en el caso base. Mi intuición me dice que el caso base es 0 o menos. Si intentamos replicar algo 0 o menos veces, debemos devolver una lista vacía. También para números negativos ya que no tiene sentido.

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

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

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

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

El primer patrón indica que si queremos obtener 0 o un número negativo de elementos, obtenemos una lista vacía. Fíjate que estamos usando _ para enlazar la lista ya que realmente no nos importa en este patrón.

Además también estamos usando una guarda, pero sin la parte **otherwise**. Esto significa que si **n** acaba siendo algo más que 0, el patrón fallará y continuará hacia el siguiente. El segundo patrón indica que si intentamos tomar algo de una lista vacía, obtenemos una lista vacía. El tercer patrón rompe la lista en cabeza y cola. Luego decimos que si tomamos **n** elementos de una lista es igual a una lista que tiene **x** como cabeza y como cola una lista que tome n-1 elementos de la cola. Intenta usar papel y lápiz para seguir el desarrollo de como sería la evaluación de **take 3 [4,3,2,1]**, por ejemplo.

reverse simplemente pone al revés una lista. Piensa en el caso base, ¿cuál es? Veamos... ¡Es una lista vacía! Una lista vacía inversa es igual a esa misma lista vacía. ¿qué hay del resto de la lista?

Podríamos decir que si dividimos una lista en su cabeza y cola, la lista inversa es igual a la cola invertida más luego la cabeza al final.

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

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

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

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

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

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

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

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

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

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

¡Quicksort!

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

Bueno, la declaración de tipo será **quicksort :: (Ord a) => [a] -> [a]**. Ninguna sorpresa. ¿Caso base? La lista vacía, como era de esperar. Ahora viene el algoritmo principal: una lista ordenada es una lista que tiene todos los elementos menores (o iguales) que la cabeza al principio (y esos valores están ordenados), luego viene la cabeza de la lista que estará en el medio y luego vienen los elementos que son mayores que la cabeza (que también estarán ordenados). Hemos dicho dos veces "ordenados", así que probablemente tendremos que hacer dos llamadas recursivas. También hemos usado dos veces el verbo "es" para definir el algoritmo en lugar de "hace esto", "hace aquello", "entonces hace"... ¡Esa es la belleza de la programación funcional! ¿Cómo vamos a conseguir filtrar los elementos que son mayores y menores que la cabeza de la lista? Con listas intensionales. Así que empecemos y definamos esta función:

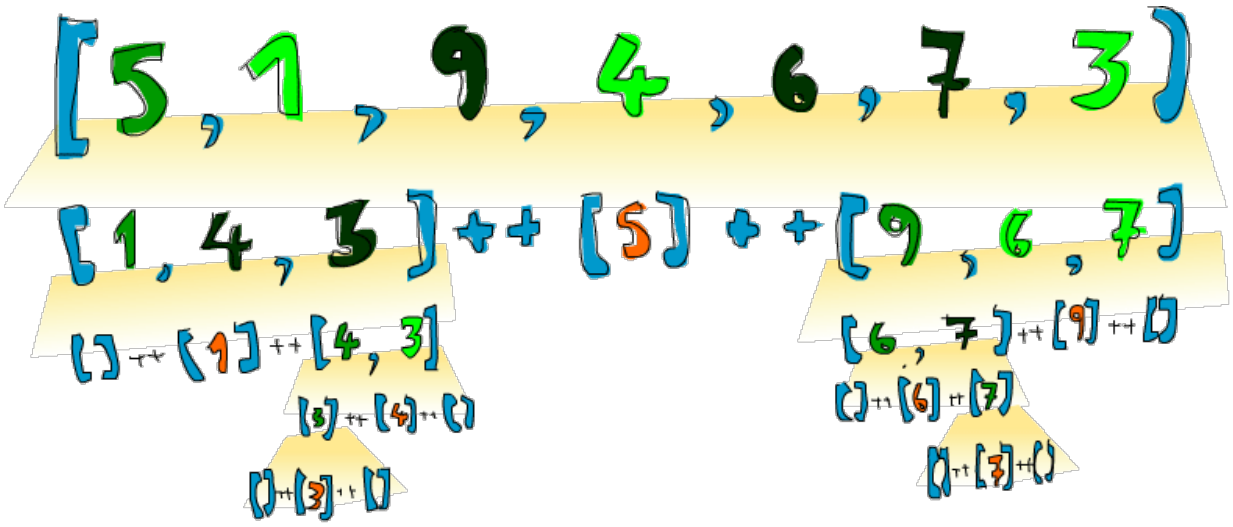
```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
   let smallerSorted = quicksort [a | a <- xs, a <= x]
       biggerSorted  = quicksort [a | a <- xs, a > x]
   in  smallerSorted ++ [x] ++ biggerSorted
```

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

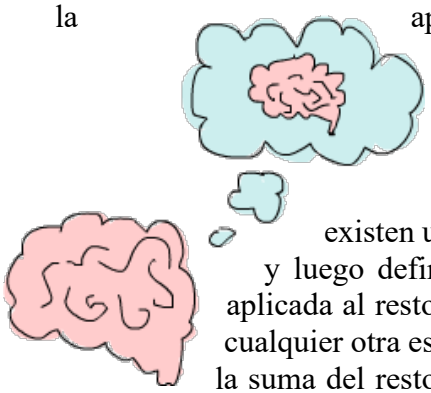
```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "el veloz murcielago hindu comia feliz cardillo y kiwi"
"        aaacccddeeeefghiiiiiiiklllllllmmnooooorruuvwyzz"
```

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

en el que alcanzaremos listas vacías y las listas vacías ya están ordenadas de alguna forma. Aquí tienes una ilustración:



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



Pensando de forma recursiva

Hemos usado un poco la recursión y como te habrás dado cuenta existen unos pasos comunes. Normalmente primero definimos los casos base y luego definimos una función que hace algo entre un elemento y la función aplicada al resto de elementos. No importa si este elemento es una lista, un árbol o cualquier otra estructura de datos. Un sumatorio es la suma del primer elemento más la suma del resto de elementos. Un productorio es el producto del primer elemento entre el producto del resto de elementos. El tamaño de una lista es 1 más el tamaño del resto de la lista, etc.

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

Es similar cuando tratamos con números. Normalmente hacemos algo con un número y luego aplicamos la función a ese número modificado. Ya hicimos funciones recursivas de este tipo como el del factorial de un número, el cual no tiene sentido con cero, ya que el factorial solo está definido para enteros positivos. A menudo el caso base resulta ser la identidad. La identidad de la multiplicación es 1 ya que si multiplicas algo por 1 obtienes el mismo resultado. También cuando

realizamos sumatorios de listas, definimos como 0 al sumatorio de una lista vacía, ya que 0 es la identidad de la suma. En Quicksort, el caso base es la lista vacía y la identidad es también la lista vacía, ya que si añades a una lista la lista vacía obtienes la misma lista ordenada.

Cargando...

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

Modelo de Programación lógico.

Introducción al modelo de programación lógico

Lógica: La lógica estudia los procesos del pensamiento para descubrir los elementos racionales que la constituyen y las funciones que los enlazan. Igualmente, la lógica indaga las relaciones mutuas y las influencias recíprocas que existen entre el pensamiento y la realidad representada por este pensamiento.

La Lógica de Primer Orden: conocida también como Lógica de Predicados, permite distinguir entre sujeto y predicado, haciendo así posible el estudio del contenido de los enunciados, algo que no era posible con la Lógica de Proposiciones.

La lógica proposicional es declarativa en este sentido, las proposiciones representan hechos que se dan o no en la realidad. La lógica de primer orden tie- Hechos nen un compromiso ontólogico más fuerte [107], donde la realidad implica además, objetos y relaciones entre ellos. Consideren los siguientes ejemplos de enunciados Objetos y relaciones declarativos:

1. Julia es madre y Luis es hijo de Julia.
2. Toda madre ama a sus hijos.

donde el enunciado (1) se refiere a los objetos de discurso Julia y Luis, usando propiedades de estos objetos, como ser madre; así como relaciones entre éstos, como ser hijo de alguien. El enunciado (2) se refiere a relaciones que aplican a todas las madres, en tanto que objetos de discurso. A esto nos referimos cuando hablamos de representación de conocimiento en el contexto de la Programación Lógica, a Representación describir una situación en términos de objetos y relaciones entre ellos. Al igual que en el caso proposicional, si se aplican ciertas reglas de prueba a tales representaciones, es posible obtener nuevas conclusiones. Por ejemplo, conociendo (1) y (2) es posible inferir (vía una versión de Modus Ponens, un poco distinta de la Inferencia proposicional) que: 3. Julia ama a Luis. De forma que, sería deseable poder describir los objetos que conforman un uni- Universo de discurso verso de discurso, personas en el ejemplo; así como las relaciones que se dan entre ellos, hijo y madre en el ejemplo; y computar tales descripciones para obtener conclusiones como (3). Al describir el problema que queremos resolver, también podemos hacer uso de funciones, relaciones en las cuales uno o más objetos del universo de Funciones discurso se relacionan con un objeto único.

Unificación y resolución

El Método de Resolución es un intento de mecanizar el proceso de deducción natural de forma eficiente.

Las demostraciones se consiguen utilizando el método refutativo (reducción al absurdo), es decir lo que se intenta es encontrar contradicciones. Para probar una sentencia basta con demostrar que su negación nos lleva a una contradicción con las sentencias conocidas (es insatisfactible).

algoritmo de resolución

Existen distintas Estrategias de Resolución: sistemática, con conjunto soporte, unitaria, primaria y lineal.

El procedimiento de resolución consiste en un proceso iterativo en el cual comparamos (resolvemos), dos cláusulas llamadas cláusulas padres y producimos una nueva cláusula que se ha inferido (deducido), de ellas.

Ejemplo

La resolución opera tomando dos cláusulas tales que cada una contenga un mismo literal, en una cláusula en forma positiva y en la otra en forma negativa. El resolvente se obtiene combinando todos los literales de las cláusulas padres y eliminando aquellos que se cancelan.

Algoritmo de unificación

Podemos definir la Unificación como un procedimiento de emparejamiento que compara dos literales y descubre si existe un conjunto de sustituciones que los haga idénticos.

Ejemplo

- Se unificara P(x, x) con P(y, z):

  Primera sustitución: (y/x)
- Resultado: P(y, y) P(y, z)

  Segunda sustitución: (z/y)
- Resultado: P(z, z) P (z, z)

La sustitución resultante es la composición de las sustituciones: s = { z/y , y/x}

Cláusulas de Horn, resolución SLD.

En lógica proposicional, una fórmula lógica es una cláusula de Horn si es una cláusula (disyunción de literales) con, como máximo, un literal positivo. Se llaman así por el lógico Alfred Horn, el primero en señalar la importancia de estas cláusulas en 1951.

Una cláusula de Horn con exactamente un literal positivo es una cláusula "definite"; en álgebra universal las cláusulas "definites" resultan como cuasi-identidades. Una cláusula de Horn sin ningún literal positivo es a veces llamada cláusula objetivo (goal) o consulta (query), especialmente en programación lógica.

Ejemplo

Se llaman cláusulas de Horn aquellas que tienen como máximo un literal positivo. Hay dos tipos:

Las cláusulas determinadas (definite clauses), o «cláusulas de Horn con cabeza» son las que sólo tienen un literal positivo:

(¬p1 ¬p2 ... ¬pk q) (p1 p2 ... pk q)

Caso particular son las no tienen más que ese literal positivo, que representan «hechos», es decir, conocimiento factual.

Los objetivos determinados (definite goals), o «cláusulas de Horn sin cabeza» son las que no tienen ningún literal positivo:

(¬p1 ¬p2 ... ¬pk) ¬(p1 p2 ... pk)

Resolución sld

La resolución general es un mecanismo muy potente de demostración pero tiene un alto grado de indeterminismo: en la selección de las clausulas con las que hacer resolución y en la selección de los literales a utilizar en la resolución.

Los hechos y las reglas se denominan clausulas definidas: Los hechos representan “hechos acerca de los objetos” (de nuestro universo de discurso), relaciones elementales entre estos objetos las reglas expresan relaciones condicionales entre los objetos, dependencias.

Reglas

Un hecho es una regla con cuerpo vacío un objetivo es una regla con cabeza vacía y el éxito es una regla con cabeza y cuerpo vacíos. En las cláusulas de Horn se trabaja con secuencias de literales en vez de conjuntos Esto implica dos cosas: los literales pueden aparecer repetidos en el cuerpo hay un orden en los literales del cuerpo.

Programación lógica con cláusulas de Horn.

La aplicación de refutación por resolución en cláusulas de Horn es un mecanismo ampliamente utilizado. El lenguaje de programación Prolog, se basa en este tipo de cláusulas y los programas implementados en él se denominan programas lógicos definidos

Tenemos tres tipos de cláusulas de Horn

Tipo I: un átomo simple (hecho) ej. P11(x)

Tipo II: una implicación (llamada regla) cuyo antecedente consiste de una conjunción de literales positivos y el consecuente es sólo un literal positivo: L1^L2…^Ln-1 Ln donde las L son literales positivos. Muchas veces se nota: L1,L2,…,Ln-1 Ln .

Tipo III: Un conjunto de literales negados, que puede notarse como una implicación sin consecuente L1, L2…Ln

La notación utilizando implicación es la preferida para escribir cláusulas de Horn y resulta equivalente a la notación utilizando la disyunción de literales. Una cláusula de Horn notada como disyunción finita de literales, siendo las L literales, se escribe así:

-L1v-L2v….-Ln-1 vLn ,
Lo cual pude notarse como conjunto:
{-L1,-L2,…..-Ln-1, Ln}
Y es equivalente a:
L1,L2,…,Ln-1 Ln (las comas son conjunciones)

En general cualquier cláusula puede escribirse como implicación, dando lugar a lo que se conoce como forma normal conjuntiva. La equivalencia es directa, si tenemos una cláusula de la forma A1 v….vAn v-B1 v….v-Bn equivale a

B1^...^Bn A1,v…v An