Entonces, ¿qué es Haskell?

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

consecuencias: si una función es llamada dos veces con los mismos parámetros, obtendremos siempre el mismo resultado. A esto lo llamamos *transparencia referencial* y no solo permite al compilador razonar acerca de el comportamiento de un programa, sino que también nos permite deducir fácilmente (e incluso demostrar) que una función es correcta y así poder construir funciones más complejas uniendo funciones simples.

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

lista y solo cuando lo necesitamos. Cuando queremos calcular algo a partir de unos datos iniciales en un lenguaje perezoso, solo tenemos que tomar estos datos e ir transformándolos y moldeándolos hasta que se parezcan al resultado que deseamos.

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

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

Haskell fue creado por unos tipos muy inteligentes (todos ellos con sus respectivos doctorados). El proyecto de crear

Haskell comenzó en 1987 cuando un comité de investigadores se pusieron de acuerdo para diseñar un lenguaje revolucionario. En el 2003 el informe Haskell fue publicado, definiendo así una versión estable del lenguaje.

Qué necesitas para comenzar

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

GHC toma un script de Haskell (normalmente tienen la extensión *.hs*) y lo compila, pero también tiene un modo interactivo el cual nos permite interactuar con dichos scripts. Podemos llamar a las funciones de los scripts que hayamos cargado y los resultados serán mostrados de forma inmediata. Para aprender es mucho más fácil y rápido en lugar de tener que compilar y ejecutar los programas una y otra vez. El modo

interactivo se ejecuta tecleando `ghci` desde tu terminal. Si hemos definido algunas funciones en un fichero llamado, digamos, `misFunciones.hs`, podemos cargar esas funciones tecleando `:l misFunciones`, siempre y cuando `misFunciones.hs` esté en el mismo directorio en el que fue invocado `ghci`. Si modificamos el script *.hs* y queremos observar los cambios tenemos que volver a ejecutar `:l misFunciones` o ejecutar `:r` que es equivalente ya que recarga el script actual. Trabajaremos definiendo algunas funciones en un fichero *.hs*, las cargamos y pasamos el rato jugando con ellas, luego modificaremos el fichero *.hs* volviendo a cargarlo y así sucesivamente. Seguiremos este proceso durante toda la guía.

Empezando

¡Preparados, listos, ya!

Muy bien ¡Vamos a empezar! Si eres esa clase de mala persona que no lee las introducciones y te la has saltado,

quizás debas leer la última sección de la introducción porque explica lo que necesitas para seguir esta guía y como vamos a trabajar. Lo primero que vamos a hacer es ejecutar GHC en modo interactivo y utilizar algunas funciones para ir acostumbrándonos un poco. Abre una terminal y escribe `ghci`. Serás recibido con un saludo como éste:

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

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

Aquí tenemos algo de aritmética simple.

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

```
2.5
ghci>
```

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

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

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

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

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

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

La comprobación de igualdad se hace así:

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

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

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

GHCi nos está diciendo es que `"texto"` no es un número y por lo tanto no sabe como sumarlo a 5. Incluso si en lugar de`"texto"` fuera `"cuatro"`, `"four"`, o `"4"`, Haskell no lo consideraría como un número. `+` espera que su parte izquierda y derecha sean números. Si intentamos realizar `True == 5`, GHCi nos diría que los tipos no coinciden. Mientras que `+`funciona solo con cosas que son consideradas números, `==` funciona con cualquiera cosa

que pueda ser comparada. El truco está en que ambas deben ser comparables entre si. No podemos comparar la velocidad con el tocino. Daremos un vistazo más detallado sobre los tipos más adelante. Nota: podemos hacer `5 + 4.0` porque `5` no posee un tipo concreto y puede actuar como un entero o como un número en coma flotante. `4.0` no puede actuar como un entero, así que `5` es el único que se puede adaptar.

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

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

Haskell, las funciones son llamadas escribiendo su nombre, un espacio y sus parámetros, separados por espacios. Para empezar, vamos a intentar llamar a una de las funciones más aburridas de Haskell.

```
ghci> succ 8
9
```

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

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

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

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

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

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

La gente que ya conoce algún lenguaje imperativo tiende a aferrarse a la idea de que los paréntesis indican una aplicación de funciones. Por ejemplo, en C, usas los paréntesis para llamar a las funciones como `foo()`, `bar(1)`, o `baz(3, "jaja")`. Como hemos dicho, los espacios son usados para la aplicación de funciones en Haskell. Así que estas funciones en Haskell serían `foo`, `bar 1` y `baz 3 "jaja"`. Si ves algo como `bar (bar 3)` no significa que `bar` es llamado con `bar` y `3`como parámetros. Significa que primero

llamamos a la función `bar` con `3` como parámetro para obtener un número y luego volver a llamar `bar` otra vez con ese número. En C, esto sería algo como `bar(bar(3))`.

Las primeras pequeñas funciones

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

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

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

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

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

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

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

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

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

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

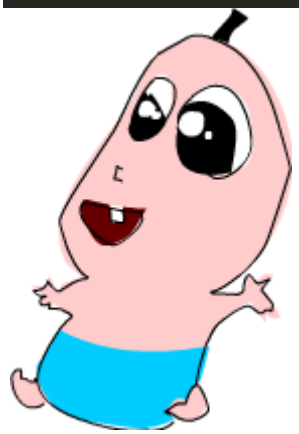
Esto es un simple ejemplo de un patrón normal que verás por todo Haskell. Crear funciones pequeñas que son obviamente correctas y luego combinarlas en funciones más complejas. De esta forma también evitarás repetirte. ¿Qué pasa si algunos matemáticos descubren que 2 es en realidad 3 y tienes que cambiar tu programa? Puedes

simplemente redefinir `doubleMe` para que sea `x + x + x` y como `doubleUs` llama a `doubleMe` automáticamente funcionara en este extraño mundo en el que 2 es 3.

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

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

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



Acabamos de introducir la sentencia `if` de Haskell. Probablemente ya estés familiarizado con la sentencia `if` de otros lenguajes. La diferencia entre la sentencia `if` de Haskell y la de los lenguajes imperativos es que la parte `else` es obligatoria. En los lenguajes imperativos podemos saltarnos unos cuantos pasos si una

condición no se ha satisfecho pero en Haskell cada expresión o función debe devolver un valor. También podríamos haber definido la sentencia `if` en una sola línea pero así parece un poco mas legible. Otro asunto acerca de la sentencia `if` en Haskell es que es una expresión. Básicamente una expresión es un trozo de código que devuelve un valor. `5` es una expresión porque devuelve 5, `4 + 8` es una expresión, `x + y` es una expresión porque devuelve la suma de `x` e `y`. Como la parte `else` es obligatoria, una sentencia `if` siempre devolverá algo y por tanto es una expresión. Si queremos sumar uno a cada número que es producido por la función anterior, podemos escribir su cuerpo así.

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

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

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

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

Una introducción a las listas

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

En Haskell, las listas son una estructura de datos **homogénea**. Almacena varios elementos del mismo tipo. Esto significa que podemos crear una lista de enteros

o una lista de caracteres, pero no podemos crear una lista que tenga unos cuantos enteros y otros cuantos caracteres. Y ahora, ¡una lista!

Nota

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

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

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

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

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

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

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

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

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

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

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

```
[1,2,3]
```

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

Nota

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

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

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

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

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

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

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

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

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

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

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

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

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

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

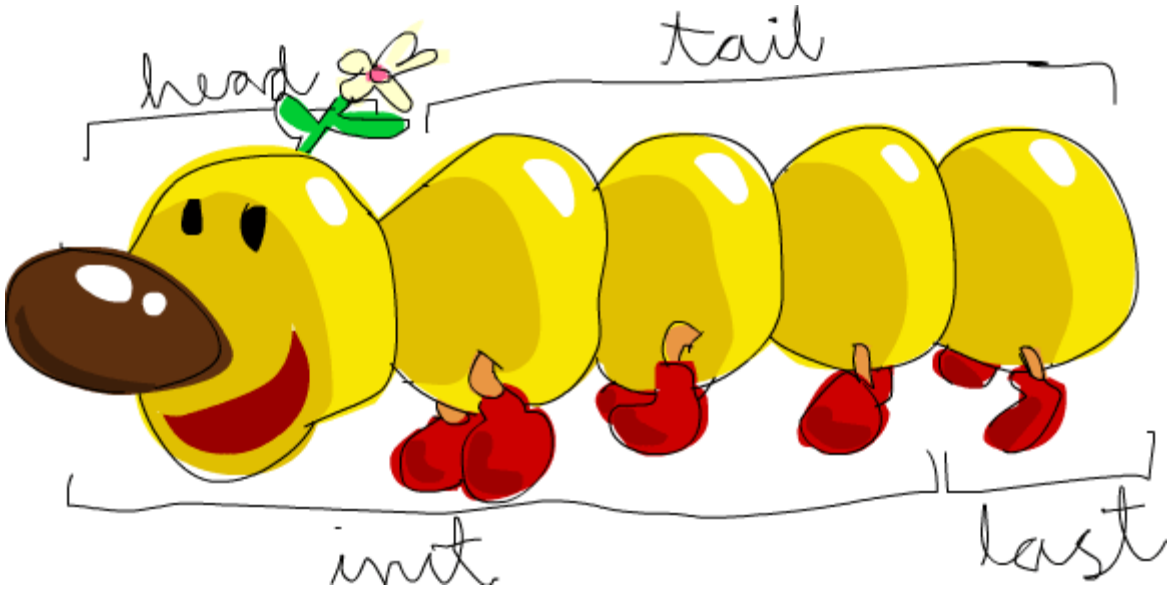
- `last` toma una lista y devuelve su último elemento.

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

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

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

Si imaginamos las listas como monstruos, serian algo como:



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

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

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

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

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

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

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

- `reverse` pone del revés una lista.

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

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

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

```
ghci> take 0 [6,6,6]
[]
```

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

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

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

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

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

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

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

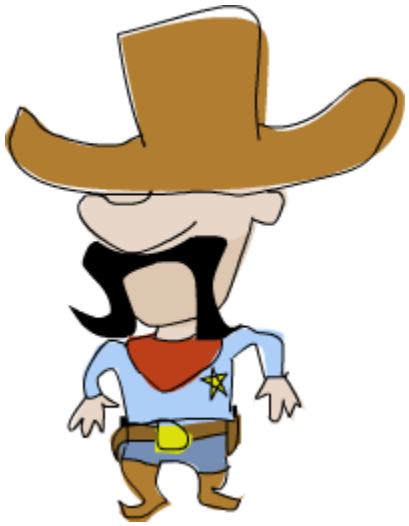
```
ghci> product [1,2,5,6,7,9,2,0]
0
```

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

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

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

Texas rangos



¿Qué pasa si queremos una lista con todos los números entre el 1 y el 20? Sí, podríamos simplemente escribirlos todos pero obviamente esto no es una solución para los que buscan buenos lenguajes de programación. En su

lugar, usaremos rangos. Los rangos son una manera de crear listas que contengan una secuencia aritmética de elementos enumerables. Los números pueden ser enumerados. Uno, dos, tres, cuatro, etc. Los caracteres también pueden ser enumerados. El alfabeto es una enumeración de caracteres desde la A hasta la Z. Los nombres no son enumerables. ¿Qué viene después de "Juan"? Ni idea.

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

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

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

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

```
ghci> [3,6..20]
[3,6,9,12,15,18]
```

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

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

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

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

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

También podemos utilizar los rangos para crear listas infinitas simplemente no indicando un límite superior. Más

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

Ahora, un par de funciones que generan listas infinitas:

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

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

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

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

Aunque aquí sería más simple usar la función `replicate`, ya que sabemos el número de

elementos de
antemano. `replicate 3 10` devuelve `[10,10,10]`.

Soy una lista intensional

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

Si quisiéramos escribir esto en Haskell, podríamos usar algo como `take 10 [2,4..]`. Pero, ¿y si no quisiéramos los dobles de los diez primeros número naturales, sino algo más complejo? Para ello podemos utilizar listas intensionales. Las listas intensionales son muy similares a los conjuntos definidos de forma intensiva. En este caso, la

lista intensional que deberíamos usar sería `[x*2 | x <- [1..10]]`. `x` es extraído de `[1..10]` y para cada elemento de `[1..10]` (que hemos ligado a `x`) calculamos su doble. Su resultado es:

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

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

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

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

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

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

intensional dentro de una función para que sea fácilmente reutilizable.

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

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

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

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

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

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

de todas las combinaciones posibles entre ambas, podemos usar algo como:

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

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

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

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

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

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

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

_ significa que no nos importa lo que vayamos a extraer de la lista, así que en vez de escribir el nombre de una variable que nunca usaríamos, simplemente escribimos _.

La función reemplaza cada elemento de la lista original por 1 y luego los suma. Esto significa que la suma resultante será el tamaño de nuestra lista.

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

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

Unas pruebas rápidas:

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

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

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

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

Tuplas

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

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

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

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

```
In the definition of `it': it = [(1, 2), (8,
11, 5), (4, 5)]
```

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

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

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

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

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

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

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

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

Nota

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

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

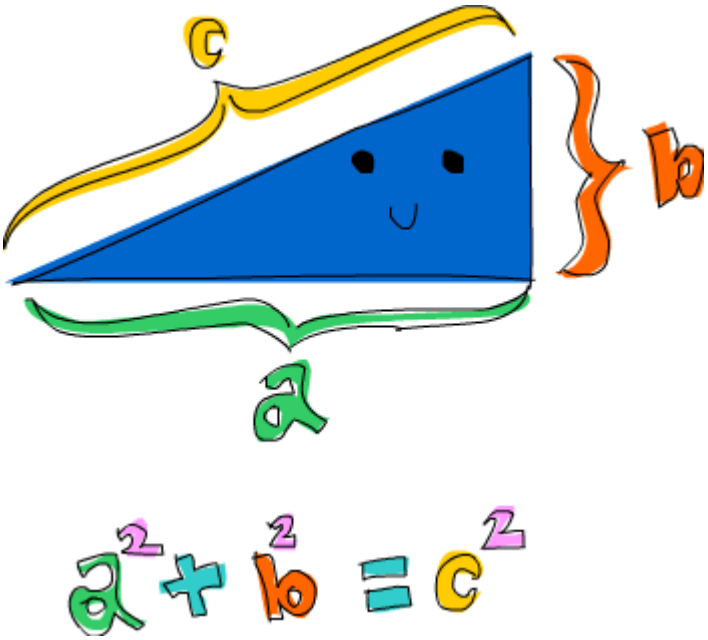
```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1 .. 5]
["uno","dos","tres","cuatro","cinco"]
[(1,"uno"),(2,"dos"),(3,"tres"),(4,"cuatro")
,(5,"cinco")]
```

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

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

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

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



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

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

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

largo que la hipotenusa y que el lado a no es más largo que el lado b.

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

¡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.