I really believe all developers should learn **Haskell**. I don't think everyone needs to be super **Haskell ninja**, but they should at least discover what **Haskell** has to offer. **Learning Haskell opens your mind**.

Mainstream languages share the same foundations:

- **variables**
- **loops**
- **Pointers**
- **data structures, objects and classes** (for most)

Haskell is very different. The language uses a lot of concepts I had never heard about before. **Many of those concepts will help you become a better programmer**.

But learning **Haskell** can be hard. It was for me. In this article I try to provide what I lacked during my learning.

This article will certainly be hard to follow. This is on purpose. There is no shortcut to learning **Haskell**. It is hard and challenging. But I believe this is a good thing. It is because it is hard that **Haskell** is interesting.

The conventional method to **learning Haskell** is to read two books. **First "Learn You a Haskell**" and just after **"Real World Haskell".** I also believe this is the right way to go. But to learn what **Haskell** is all about, you'll have to read them in detail.

In contrast, this article is a very brief and dense overview of all major aspects of **Haskell**. I also added some information I lacked while I learned **Haskell**.

The tutorial contains five parts:

- **Introduction:** a short example to show Haskell can be friendly
- **Basic Haskell: Haskell** syntax, and some essential notions
- **Hard Difficulty Part:**
    - **Functional style**; a progressive example, from imperative to functional style
    - **Types**; types and a standard binary tree example
    - **Infinite Structure**; manipulate an infinite binary tree!
- **Hell Difficulty Part:**
    - **Deal with IO**; A very minimal example
    - **IO trick explained**; the hidden detail I lacked to understand IO
    - **Monads**; incredible how we can generalize
- **Appendix:**
    - **More on infinite tree**; a more math oriented discussion about infinite trees

Note: Each time you see a separator with a filename ending in .lhs you can click the filename to get this file. If you save the file as filename.lhs, you can run it with **runhaskell filename.lhs**

Introduction

Install

- ***Haskell Platform*** is the standard way to install **Haskell**.

Tools

- **ghc**: Compiler similar to gcc for C.
- **ghci**: Interactive **Haskell** **(REPL)**
- **runhaskell**: Execute a program **without compiling it**. Convenient but very slow compared to compiled programs.

Don't be afraid

Many books/articles about **Haskell** start by introducing some esoteric formula **(quicksort, Fibonacci, etc...)**. I will do the exact opposite. At first I won't show you any **Haskell** **super power**. I will start with similarities between **Haskell** and other programming languages. Let's jump to the mandatory **"Hello World"**.

```
main = putStrLn "Hello World!"
```

To run it, you can save this code in a **01helloWorld.hs** and:

```
$ runhaskell ./01helloWorld.hs
Hello World!
```
enter

You could also download the **literate** **Haskell** **source**. You should see a link just above the introduction title.

```
> main = putStrLn "Hello World!"
```

Save this code in a file named **02helloWorld.lhs**

```
$ runhaskell ./02helloWorld.lhs
Hello World!
```
enter

Now, the program will ask you a name and replying **"Hello"** and using the name you entered:

```
--03hello.hs
main = do
    print "What is your name?"
```

```
    name <- getLine
    print ("Hello " ++ name ++ "!")
```

```
$ runghc ./03hello.hs          enter
"What is your name?"
Diana                          enter
"Hello Diana!"
```

First, let us compare this with similar programs in a few **imperative languages**:

```
# Python
```

```
print "What is your name?"
name = raw_input()
print "Hello %S!" % name
```

```
# Ruby
```

```
puts "What is your name?"
name = gets.chomp
puts "Hello #{name}!"
```

```
// In C
```

```
#include <stdio.h>
int main (int argc, char **argv) {
    char name[666]; // <- An Evil Number!
    // What if my name is more than 665 character long?
    printf("What is your name?\n");
    scanf("%s", name);
    printf("Hello %s!\n", name);
    return 0;
}
```

The **structure is the same**, but there are **some syntax differences**. The main part of this tutorial will be dedicated to explaining why.

Haskell has a **main function** and **every object has a type**. The **type of main** is **IO ()**. **This means main will cause side effects**. Just remember that Haskell looks a like **mainstream imperative languages**.

Very basic Haskell

Before continuing you need to be warned about some essential **Haskell** properties.

Haskell properties

Functional

Haskell is a **functional language**. If you have an **imperative language** background, you'll have to learn a lot of new things. Hopefully many of these new concepts will help you to **program even with imperative languages**.

Smart Static Typing

Instead of being not a friendly way like in **C, C++ or Java**, the **Haskell type system** is here to help you.

Purity

Pure functions won't modify anything in the outside world. This means they can't modify the **value of a variable**, can't get user input, can't write on the screen, can't launch a missile. On the other hand, **parallelism** will be very **easy to achieve**. **Haskell** makes it clear where **effects occur** and where your **code is pure**. Also, it will be far easier to reason about your program. Most bugs will be prevented in the **pure parts of your program**.

Furthermore, **pure functions** follow a **fundamental law** in **Haskell**:

Applying a function the same parameters always returns the same value

Laziness

Laziness by default is a very uncommon language design. By default, **Haskell evaluates something only when it is needed**. In consequence, it provides a very elegant way to manipulate **infinite structures**, for example.

A last warning about how you should read **Haskell code**. In my opinion, it's like reading scientific papers. Some parts are very clear, but when you see an expression, just focus and read slower. Also, while learning **Haskell**, it **really** doesn't matter how much do you understand **syntax** details. If you don't know what means a symbol like **>>=**, **<$>**, **<-** or any other **weird symbol**, just ignore it and follows the flow of the code.

Function declaration

You might be used to declaring functions like this:

C

```
int f(int x, int y) {
    return x*x + y*y;
}
```

JavaScript

```
function f(x,y) {
    return x*x + y*y;
}
```

Python

```
def f(x,y):
    return x*x + y*y
```

Ruby

```
def f(x,y)
    x*x + y*y
end
```

Schem

```
(define (f x y)
    (+ (* x x) (* y y)))
```

Haskell way is

```
f x y = x * x + y * y
```

Very clean. No parenthesis, no def.

Don't forget, **Haskell** uses **functions** and **types** a lot. It is thus very easy to define them. The syntax was particularly well thought out for these objects.

A Type Example

Although it's not mandatory, **function type signature** is usually made in a **explicit** way. It's not mandatory because the **compiler is so smart** enough to figure out it for you. But, writing a function type signature is a good idea because indicates intent and understanding.

Let's play a little. We declare the **type** using **(: :)**

```haskell
--04operacion.hs
foo :: Int -> Int -> Int
foo x y = x * x + y * y
main = print $ foo 2 3
```

```
$ runhaskell 04operacion.hs                                    enter
13
```

Now try

```haskell
--05operacion.hs
foo :: Int -> Int -> Int
foo x y = x * x + y * y
main = print (foo 2.3 4.2)
```

```
$ runhaskell 05operación.hs                                    enter
21_very_basic.lhs:6:23:
    No instance for (Fractional Int)
      arising from the literal `4.2'
Possible fix: add an instance declaration for (Fractional Int)
  In the second argument of `foo', namely `4.2'
  In the first argument of `print', namely `(foo 2.3 4.2)'
  In the expression: print (foo 2.3 4.2)
```

You have got an **error**, because **4.2 isn't an Int**.
The solution: don't declare a **type** for **foo function** at this moment and let **Haskell infer** the most general **type**:

```haskell
--06operacion.hs
foo x y = x * x + y * y
main = print (foo 2.3 4.2)
```

```
$ runhaskell operación3.hs                                     enter
22.93
```

It works! Luckily, we don't have to declare a new **function** for **every single type**. For example, in **C**, **you'll have to declare a function for int, for float, for long, for double, etc...**

But, what type should we declare? To discover the **type** **Haskell** has found for us, just launch **ghci:**

```
$ ghci                                                          enter
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude> :l 06operacion.hs                                      enter
[1 of 1] Compiling Main       ( 06operacion.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t foo                                                   enter
foo :: Num a => a -> a -> a
```

Hmmmm.... **What is this strange type?**

```
Num a => a -> a -> a
```

First, let's focus on the right part `a -> a -> a`. To understand it, just look at a list of progressive examples:

The written type	Its meaning
Int	***the type Int***
Int -> Int	***the type function from Int to Int***
Float -> Int	***the type function from Float to Int***
a -> Int	***the type function from any type to Int***
a -> a	***the type function from any type a to the same type a***
a -> a -> a	***the type function of two arguments of any type a to the same type a***

In the **type a -> a -> a**, the letter **a** is a **type variable**. It means f is a **function** with **two arguments** and both arguments and the result have the **same type**. The **type variable a** could take many different **type values**. For example **Int, Integer, Float**...

So instead of having a **forced type** like in **C** and having to **declare a function** for **int, long, float, double, etc.**, we declare only one **function** like in a **dynamically typed language**.

This is sometimes called **parametric polymorphism**. It's also called having your cake and eating it too.

Generally **a** can be any **type**, for example a **String** or an **Int**, but also more complex **types**, like **Trees**, other **functions**, etc. But here our **type** is prefixed with **Num a =>**.

Num is a type class. **A type class can be understood as a set of types**. **Type class Num** contains only **types** which behave like **numbers**. More precisely, **Num is class** containing **types** which implement a **specific list of functions (also called methods)**, and in particular **(+)** and **(*)**.

Type classes are a very **powerful language construct**. We can do some incredibly powerful stuff with this. Later, will be returning on this matter.

Finally, **Num a => a -> a -> a** means:

The **type a** is belonging to the **Num typeclass**. It's a **function** from **type a to (a -> a)**.

Yes, strange. In fact, **Haskell** **functions** really haven't **two arguments**. Instead all **functions** have only **one argument**. But we will note that **taking two arguments** is equivalent to **take one argument** and **returning** a **function taking the second argument as a parameter**.

More precisely **f 3 4** is equivalent to **(f 3) 4**. Note **f 3** is a **function**:

```
f :: Num a => a -> a -> a
g :: Num a => a -> a
g = f 3
g y ⇔ 3 * 3 + y * y
```

Another notation exists for **functions**. The **lambda notation** allows us to create **functions** without **assigning them a name**. We call them **anonymous functions**. We could also have written:

```
g = \y -> 3 * 3 + y * y
```

The **** is used because it looks **like** λ and is **ASCII**.

If you are not used to **functional programming** your brain should be starting to heat up. It is time to make a real application.

But just before that, we should verify the **type system** works as expected:

```
f :: Num a => a -> a -> a
f x y = x * x + y * y
main = print $ f 3 2.4
```

It works, because, **3** is a valid representation both for **Fractional numbers** like **Float** and for **Integer**. As **2.4** is a **Fractional number**, **3** is then interpreted as being also a **Fractional number**.

If we force our **function** to work with different **types**, it will fail:

```
--07operacion.hs
f :: Num a => a -> a -> a
f x y = x * x + y * y
x :: Int
x = 3
y :: Float
y = 2.4
main = print (f x y)
```

```
Prelude> :l 07operacion.hs
[1 of 1] Compiling Main    ( 07operacion.hs, interpreted )

operacion4.hs:14:19:
 Couldn't match expected type ‘Int’ with actual type ‘Float’
 In the second argument of ‘f’, namely ‘y’
 In the first argument of ‘print’, namely ‘(f x y)’
Failed, modules loaded: none.
```

enter

won't work because **type x ≠ type y**

The **compiler complains**. The **two parameters** must have the **same type**.

If you believe that this is a bad idea, and that the **compiler** should make the transformation from one type to another for you, you should really watch this great (and funny) video: **WAT**

Check this program

```
--08operacion.hs
--pure function
operacion :: Num a => a -> a -> a
operacion x y = x * x + y * y
--main program
main = do
  putStrLn "ingresa un numero:"
  x <- getLine
  putStrLn "ingresa otro numero:"
```

```
  y <- getLine
  print $ operacion (read x) (read y)
```

```
$ runghc ./08operacion.hs                enter
ingresa un numero:
8                                        enter
ingresa otro numero:
7                                        enter
113
```

Essential Haskell

I suggest that you skim this part. Think of it as a reference. **Haskell** has a lot of features. A lot of information is missing here. Come back here if the notation feels strange.

I use the ⇔ symbol to state that **two expression are equivalent**. It's a meta notation, ⇔ doesn't exists in **Haskell**. I will also use ⇒ to show what the return value of an **expression** is.

Notations

Arithmetic

```
3 + 2 * 6 / 3 ⇔ 3 + ((2 * 6) / 3)
```

Logic

```
True || False ⇒ True
True && False ⇒ False
True == False ⇒ False
True /= False ⇒ True  (/=) is the operator for not equal
```

Powers

```
x ^ n     for n an integral (Int or Integer)
x ** y    for y any kind of number (Float for example)
```

Integer has no limit except your computer capacity:

```
Prelude> 4 ^ 103                                                  enter
102844034832575377634685573909834406561420991602098741459288064
```

Yeah! And also **rational numbers** FTW! But you need to import the **module Data.Ratio**:

```
Prelude>:m Data.Ratio                                  enter
Data.Ratio> (11 % 15) * (5 % 3)                        enter
11 % 9
```

Lists

```
[]                      ⇔ empty list
[1,2,3]                 ⇔ List of integral
["foo","bar","baz"]     ⇔ List of String
1:[2,3]                 ⇔ [1,2,3], (:) prepend one element
1:2:[]                  ⇔ [1,2]
[1,2] ++ [3,4]          ⇔ [1,2,3,4], (++) concatenate
[1,2,3] ++ ["foo"]      ⇔ ERROR: String ≠ Integral
[1..4]                  ⇔ [1,2,3,4]
[1,3..10]               ⇔ [1,3,5,7,9]
[2,3,5,7,11..100]       ⇔ ERROR: I am not so smart!
[10,9..1]               ⇔ [10,9,8,7,6,5,4,3,2,1]
```

Strings

In **Haskell** **strings** are **list of Char**.

```
'a' :: Char
"a" :: [Char]
""  ⇔ []
"ab" ⇔ ['a','b'] ⇔  'a':"b" ⇔ 'a':['b'] ⇔ 'a':'b':[]
"abc" ⇔ "ab"++"c"
```

Remark: *In **real code** you shouldn't use **list of char** to **represent text**. You should mostly use **Data.Text** instead. If you want to represent a **stream** of **ASCII char**, you should use **Data.ByteString**.*

Tuples

The **type of couple** is **(a,b)**. **Elements** in a **tuple** can have **different types**.

```
All these tuples are valid
(2,"foo")
(3,'a',[2,3])
((2,"a"),"c",3)
```

```
fst (x,y)        ⇒  x
snd (x,y)        ⇒  y

fst (x,y,z)      ⇒  ERROR: fst :: (a,b) -> a
snd (x,y,z)      ⇒  ERROR: snd :: (a,b) -> b
```

Deal with parentheses

To remove some **parentheses** you can use **two functions**: **($)** and **(.)**.

```
By default:
f g h x          ⇔  (((f g) h) x)

-- the $ replace parenthesis, from the $
-- to the end of the expression
f g $ h x        ⇔  f g (h x) ⇔ (f g) (h x)
f $ g h x        ⇔  f (g h x) ⇔ f ((g h) x)
f $ g $ h x      ⇔  f (g (h x))

-- (.) the composition function
(f . g) x        ⇔  f (g x) ⇔ f . g $ x
(f . g . h) x    ⇔  f (g (h x)) ⇔ f . g. h $ x
```

Useful **notations** for **functions**

```
Just a reminder:
x :: Int              ⇔ x is the type Int
x :: a                ⇔ x can be of any type
x :: Num a => a       ⇔ x can be any type a
                        such that a belongs to Num typeclass
f :: a -> b           ⇔ f is a function from a to b
f :: a -> b -> c      ⇔ f is a function from a to (b → c)
f :: (a -> b) -> c    ⇔ f is a function from (a → b) to c
```

Remember that **defining** the **function type signatures** before its declaration isn't mandatory. **Haskell infers** the most general **type** for you. But it is considered a good practice to do so.

Infix notation

```
square :: Num a => a -> a
square x = x ^ 2
```

Note **^** uses **infix notation**. For each **infix operator** there its associated **prefix notation**. You just have to put it **inside parenthesis**.

```
square' x = (^) x 2
square'' x = (^2) x
```

We can **remove x** in the **left** and **right** side! It's called η**-reduction**.

```
square''' = (^2)
```

Note we can declare **functions** with **(')** in their name. Here:

square ***square'*** ***square''*** ***square'''***

Tests

An implementation of the **absolute function**.

```
absolute :: (Ord a, Num a) => a -> a
absolute x = if x >= 0 then x else -x
```

Note: the **if .. then .. else Haskell** notation is more like the **¤?¤:¤ C operator**. You cannot forget the else.

Another equivalent version:

```
absolute' x
  | x >= 0    = x
  | otherwise = -x
```

Notation warning: *indentation is important in* ***Haskell****. Like in* ***Python****, bad indentation can break your code!*

```
--09absoluto.hs
--utilizando if-then-else
absoluto :: (Ord a, Num a) => a -> a
absoluto x = if x >= 0 then x else -x
--utilizando guardas
absoluto' :: (Ord a, Num a) => a -> a
absoluto' x
```

```
  | x >= 0    = x
  | otherwise = -x
```

```
Prelude> :l 09absoluto                                          enter
[1 of 1] Compiling Main      ( 09absoluto.hs, interpreted )
Ok, modules loaded: Main.
*Main> absoluto (-3)                                            enter
3
*Main> absoluto' (-3)                                           enter
3
```

Hard Part

The **hard part** is now beginning.

Functional style

In this section, I will give a short example of the **impressive refactoring** ability provided by **Haskell**. We will **select a problem** and **solve it** in a **standard imperative way**. Then I will make the code evolve. The end result will be both more elegant and easier to adapt.

Let's solve the following problem:

Given a list of integers, return the sum of the even numbers in the list.

`example: [1,2,3,4,5] ⇒ 2 + 4 ⇒ 6`

Showing differences between **functional** and **imperative** approaches, I'll start by providing an **imperative solution** **(JavaScript)**:

```
function evenSum(list) {
    var result = 0;
    for (var i=0; i< list.length ; i++) {
        if (list[i] % 2 ==0) {
            result += list[i];
        }
    }
    return result;
}
```

Haskell, by contrast, doesn't have **variables** or **for loop iterations**. The good way to achieve the same result without **loops** is to use **recursion**.

Remark: Recursion, in imperative languages, has been perceived as a very slow execution. But this isn't the case in functional programming. Most of the time Haskell will handle recursive functions efficiently.

Here is a **recursive function** in **C language**. Note that for simplicity I assume the **int list** ends with the **first 0 value**.

```
int evenSum(int *list) {
    return accumSum(0,list);  }
int accumSum(int n, int *list) {
    int x;
    int *xs;
    if (*list == 0) { // if the list is empty
        return n;
    } else {
        x = list[0]; // let x be the first element of the list
        xs = list+1; // let xs be the list without x
        if ( 0 == (x%2) ) { // if x is even
            return accumSum(n+x, xs);
        } else {
            return accumSum(n, xs);
        }
    }
}
```

Keep this code in mind. We will translate it into **Haskell**. First, however, I need to introduce **three simple** but useful **functions** we will use:

```
even :: Integral a => a -> Bool
head :: [a] -> a
tail :: [a] -> [a]
```

Even function verifies if a **number** is **even**.

```
even :: Integral a => a -> Bool
even 3  ⇒ False
even 2  ⇒ True
```

head function returns a **list first element**:

```
head :: [a] -> a
head [1,2,3] ⇒ 1
head []      ⇒ ERROR
```

Tail function returns a **list** with **all elements**, except the **first**:

```
tail :: [a] -> [a]
tail [1,2,3] ⇒ [2,3]
tail [3]     ⇒ []
tail []      ⇒ ERROR
```

Note that for any **non empty list lst**, `lst ⇔ (head l):(tail l)`

First **Haskell** solution: **function evenSum** returns the **sum** of all **even numbers** in a **list**:

```
--10sumaPares.hs
--functions
evenSum :: [Integer] -> Integer
evenSum lst = accumSum 0 lst

accumSum :: Integral t => t -> [t] -> t
accumSum n lst = if lst == []
                   then n
                 else let x  = head lst
                          xs = tail lst
                      in if even x
                           then accumSum (n + x) xs
                         else accumSum n xs
```

```
Prelude> :l 10sumaPares                                    enter
[1 of 1] Compiling Main    ( 10sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                      enter
6
```

Here is an example, when you execute **10sumaPares.hs**:

```
*Main> evenSum [1..5]
accumSum 0 [1,2,3,4,5]
1 is odd
accumSum 0 [2,3,4,5]
2 is even
accumSum (0+2) [3,4,5]
3 is odd
accumSum (0+2) [4,5]
2 is even
accumSum (0+2+4) [5]
5 is odd
accumSum (0+2+4) []
l == []
0+2+4
0+6
6
```

Coming from an **imperative language** all should seem right. In fact, many things can be improved here. First, **we can generalize the type**.

```
evenSum :: Integral a => [a] -> a
```

Next, we can use **sub functions** using **where** or **let**. This way our **accumSum function** won't pollute the namespace of our module.

```
--11sumaPares.hs
evenSum :: Integral a => [a] -> a
evenSum lst = accumSum 0 lst

  where accumSum n lst =
          if lst == []
            then n
          else let x  = head lst
                   xs = tail lst
               in if even x
                    then accumSum (n + x) xs
                  else accumSum n xs
```

```
Prelude> :l 11sumaPares                                          enter
[1 of 1] Compiling Main      ( 11sumaPares.hs, interpreted )
```

```
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                    enter
6
```

Next, we can use **pattern matching**.

```
--12sumaPares.hs
evenSum :: Integral a => [a] -> a
evenSum lst = accumSum 0 lst
  where
    accumSum n [] = n
    accumSum n (x:xs) =
      if even x
        then accumSum (n + x) xs
        else accumSum n xs
```

```
*Main> :l 12sumaPares                                    enter
[1 of 1] Compiling Main      ( 12sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                    enter
6
```

What is **pattern matching**? Use **values** instead of general **parameter names**.

Instead writing: `foo lst = if lst == [] then <x> else <y>` write:

```
foo []  = <x>
foo lst = <y>
```

But **pattern matching** goes even further. It is also able to inspect the inner data of a **complex value**.

```
foo lst = let x  = head lst
              xs = tail lst
          in if even x
                then foo (n + x) xs
             else foo n xs
```

Replace with:

```
foo (x:xs) = if even x
               then foo (n + x) xs
             else foo n xs
```

This is a **very useful feature**. It makes our code both terser and easier to read.

In **Haskell** you can simplify **function definitions** by **η-reducing** them. For example:

instead writing:

```
f x = (some expression) x
```

only write

```
f = some expression
```

We use this **method (η-reducing)** to remove the **lst**:

```
--13sumaPares.hs
evenSum :: Integral a => [a] -> a
evenSum = accumSum 0
  where
    accumSum n [] = n
    accumSum n (x:xs) =
      if even x
        then accumSum (n + x) xs
      else accumSum n xs
```

```
*Main> :l 13sumaPares                                             enter
[1 of 1] Compiling Main     ( 13sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                             enter
6
```

Next, using **guards**:

```
--14sumaPares.hs
evenSum :: Integral a => [a] -> a
evenSum = accumSum 0
  where
    accumSum n [] = n
    accumSum n (x:xs)
      |even x     = accumSum (n + x) xs
      |otherwise = accumSum n xs
```

```
Prelude> :l 14sumaPares                                             enter
[1 of 1] Compiling Main       ( 14sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                               enter
6
```

Higher Order Functions

To make things even better we should use **higher order functions**. ***What are these beasts?*** **Higher order functions are functions taking functions as parameters**.
Here are some examples:

```
filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Let's proceed by small steps.

```
--15sumaPares.hs
evenSum :: Integral a => [a] -> a
evenSum lst = mysum 0 (filter even lst)
  where
    mysum n []     = n
    mysum n (x:xs) = mysum (n + x) xs
```

```
*Main> :l 15sumaPares                                               enter
[1 of 1] Compiling Main       ( 15sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                               enter
6
```

```
filter even [1..10] ⇔  [2,4,6,8,10]
```

Filter function takes a **function of type (a -> Bool)** and **type [a] list**. It **returns a list** containing **only elements** for which the **function returned true**.

Our next step is to use another technique to accomplish the same thing as a **loop**. We will use the **foldl function** to **accumulate a value** as we pass through the list. The **foldl function** captures a general **coding pattern**:

```
myfunc list = foo initialValue list
  foo accumulated []   = accumulated
  foo tmpValue (x:xs) = foo (bar tmpValue x) xs
```

Which can be replaced it by:

```
myfunc list = foldl bar initialValue list
```

If you really want to know how the magic works, here is the definition of **foldl**:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
foldl f z [x1,...xn] ⇔ f (... (f (f z x1) x2) ...) xn
```

But, remember **Haskell** is lazy, it **doesn't evaluate (f z x)** and simply pushes it onto the **stack**. For this reason we generally use **foldl' instead** of **foldl**; **foldl' is a strict version of foldl**. If you don't understand what **lazy** and **strict** means, don't worry, just follow the code as if **foldl** and **foldl'** were the same.

Now our new version of **evenSum function** becomes:

```
--16sumaPares.hs
import Data.List
evenSum :: Integral a => [a] -> a
evenSum lst = foldl' mysum 0 (filter even lst)
  where mysum acc value = acc + value
```

```
*Main> :l 16sumaPares                                        enter
[1 of 1] Compiling Main      ( 16sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                        enter
6
```

Remember: that foldl' isn't accessible by default it will need to be imported from Data.List module

We can also simplify this by using directly a **lambda notation**. This way we don't have to create the **temporary name mysum**.

```
--17sumaPares.hs
import Data.List (foldl')
evenSum :: Integral a => [a] -> a
evenSum lst = foldl' (\x y -> x + y) 0 (filter even lst)
```

```
*Main> :l 17sumPares                                              enter
[1 of 1] Compiling Main       ( 17sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                             enter
6
```

Remember: usually it is considered a good practice to import only function(s) that the program need it.

Of course, notice that:

`(\x y -> x + y) ⇔ (+)`

Then:

```
--18sumaPares.hs
import Data.List (foldl')
evenSum :: Integral a => [a] -> a
evenSum lst = foldl' (+) 0 $ filter even lst
```

```
*Main> :l 18sumaPares                                             enter
[1 of 1] Compiling Main       ( 18sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                             enter
6
```

foldl' isn't the easiest function to grasp. If you are not used to it, you should study it a bit.

To help you to understand what's going on here, let's look at a step by step evaluation:

```
evenSum [1,2,3,4]
⇒ foldl' (+) 0 (filter even [1,2,3,4])
⇒ foldl' (+) 0 [2,4]
⇒ foldl' (+) (0 + 2) [4]
⇒ foldl' (+) 2 [4]
⇒ foldl' (+) (2 + 4) []
⇒ foldl' (+) 6 []
⇒ 6
```

Another useful **higher order function** is **(.)**. The **(.) function** corresponds to **mathematical composition**.

`(f . g . h) x ⇔  f ( g (h x))`

We can take advantage of this operator to η**-reduce** our **function**:

```
--19sumaPares.hs
import Data.List (foldl')
evenSum :: Integral a => [a] -> a
evenSum = (foldl' (+) 0) . (filter even)
```

```
*Main> :l 19sumaPares                                          enter
[1 of 1] Compiling Main     ( 19sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                          enter
6
```

Also, we **could rename** some parts to make it clearer:

```
--20sumaPares.hs
import Data.List (foldl')
evenSum :: Integral a => [a] -> a
evenSum = suma . filter even
suma :: (Num a) => [a] -> a
suma = foldl' (+) 0
```

```
*Main> :l 20sumaPares                                          enter
[1 of 1] Compiling Main     ( 20sumaPares.hs, interpreted )
Ok, modules loaded: Main.
```

```
*Main> evenSum [1..5]                                          enter
6
```

It's time to think about the direction that our code has been moved as we introduced more **functional features**. ***What we did gain by using higher order functions?***

At first, you might think the main difference is about the style. But in fact, it has more to do with a better way to think about the program. Suppose we want to modify our **function slightly**, for example, to get the **sum** of all **even squares** of elements of the **list**.

[1,2,3,4] ▷ [1,4,9,16] ▷ [4,16] ▷ 20

Writing **21sumaPares.hs** from **20sumaPares.hs** is so easy:

```
squareEvenSum  = sum . (filter even) . (map (^2))
squareEvenSum' = evenSum . (map (^2))
```

We just had to add another **“transformation function” [^0216]**.

```
--21sumaPares.hs
--suma pares de los cuadrados de una lista
SquareEvenSum :: Integral a => [a] -> a
squareEvenSum = suma . pares . cuadrados
--suma pares de una lista de números
evenSum :: Integral a => [a] -> a
evenSum = suma . pares
cuadrados :: Integral a => [a] -> [a]
cuadrados = map (^2)
suma :: Integral a => [a] -> a
suma = foldl' (+) 0
pares :: Integral a => [a] -> [a]
pares =  filter even
```

```
Prelude> :l 21sumaPares                                        enter
[1 of 1] Compiling Main     ( 21sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                          enter
6
*Main> squareEvenSum [1..5]                                    enter
20
```

map (^2) [1,2,3,4] ⇔ [1,4,9,16]

The map function applies a function to a list of all elements

We didn't have to modify anything **inside** the **function definition**. This makes the code more **modular**. But in addition you can think more **mathematically** about your **function**. You can also use your **function** interchangeably with others, as it needed. That is, you can compose, **map, fold, filter** using your **new function**.

If you believe we have reached the end of generalization, you are wrong. For example, there is a way to not only use this **function** on **lists** but on any **recursive type**. If you want to know how, I suggest you to read this quite fun article: ***Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire by Meijer, Fokkinga and Paterson***.

This example should show you how great **pure functional programming** is. Unfortunately, using **pure functional programming** isn't well suited to all usages. Or at least such a language hasn't been found yet.

One of the great **powers of Haskell** is the ability to create **DSLs (Domain Specific Language)** making it easy to change the **programming paradigm**.

In fact, **Haskell** is also great when you want to write **imperative style programming**. Understanding this was really hard for me to grasp when first learning **Haskell**. A lot of effort tends to go into explaining the superiority of the **functional approach**. Then when you start using an **imperative style with Haskell**, it can be hard to understand when and how to use it.

```
--22sumaPares.hs
import Data.List (foldl')
--functions
--listas intencionales
evenSum :: Integral a => [a] -> a
evenSum xs = sum [x | x <- xs, even x]
--composición de funciones: sum, map y función lambda con mod
evenSum1 :: Integral a => [a] -> a
evenSum1 = sum . map (\x -> if x `mod` 2 == 0 then x else 0)
--utilizamos recursividad
evenSum2 :: Integral a => [a] -> a
evenSum2 = acumulaSum 0
  where
    acumulaSum n [] = n
    acumulaSum n (x:xs) =
```

```
      if even x
        then acumulaSum (n + x) xs
      else acumulaSum n xs
--composición de funciones: sum, filter y even
evenSum3 :: Integral a => [a] -> a
evenSum3 = sum . filter even
--composición de funciones: foldl', filter y even
evenSum4 :: Integral a => [a] -> a
evenSum4 = (foldl' (+) 0) . (filter even)
--composición de funciones: suma y pares
evenSum5 :: Integral a => [a] -> a
evenSum5 = suma . pares

suma :: Integral a => [a] -> a
suma = foldl' (+) 0

pares :: Integral a => [a] -> [a]
pares =  filter even
```

```
Prelude> :l 22sumaPares                                         enter
[1 of 1] Compiling Main     ( 22sumaPares.hs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]                                           enter
6
*Main> evenSum1 [1..5]                                          enter
6
*Main> evenSum2 [1..5]                                          enter
6
*Main> evenSum3 [1..5]                                          enter
6
*Main> evenSum4 [1..5]                                          enter
6
*Main> evenSum4 [1..5]                                          enter
6
```

But before talking about this **Haskell** **super-power**, we must talk about another essential aspect of **Haskell**: **Types**.

Types

- **type Name = AnotherType** *is just an* ***alias*** *and the* ***compiler doesn't mark any difference*** *between* ***Name*** *and* ***AnotherType***
- **data Name = NameConstructor AnotherType** *does* ***mark a difference***

- **data** *can **construct structures** which can be **recursives***
- **deriving** *is magic and **creates functions** or **methods** for you*

Haskell: **types are strong and static**.

Why is this important? It will help you greatly to avoid mistakes. **Haskell** can caught most bugs during **compilation time**. The main reason for that is the **type inference** in **compilation time**. **Type inference** makes it easy to detect where you used the **wrong parameter** at the **wrong place**.

Type inference

Static typing usually is very important for **fast execution**. But most **statically typed languages** are so bad **generalizing concepts**. **Haskell's** saving grace is that it can **infer types**.

Here is a simple example, the **Haskell square function**:

```
square x = x * x
```

Square function can **work** with all **number type**. You can provide **square** with: **Int**, **Integer**, **Float**, **Fractional** and also **Complex**. Try this example:

```
Prelude> let square x = x * x                    enter
Prelude> square 2                                enter
4
Prelude> square 2.1                              enter
4.41
Prelude> :m Data.Complex                         enter
Prelude Data.Complex> square (2 :+ 1)            enter
3.0 :+ 4.0
Prelude Data.Complex> :t square                  enter
square :: Num a => a -> a
```

`x :+ y` is the notation for the **complex (x + iy)**.

Now, match with **square function C code**:

```
int int_square(int x) { return x*x; }
float   float_square(float x) {return x*x; }
complex complex_square (complex z) {
    complex tmp;
    tmp.real = z.real * z.real - z.img * z.img;
```

```
    tmp.img = 2 * z.img * z.real;
}
complex x,y;
y = complex_square(x);
```

For each **type**, you need to write a **new function**. The only way to work around this problem is to use some **meta-programming trick**, for example using the **pre-processor**. In **C++** there is a better way, **C++ templates**:

```
#include <iostream>
#include <complex>
using namespace std;
template<typename T>
T square(T x)
{
    return x*x;
}
int main() {
    // int
    int sqr_of_five = square(5);
    cout << sqr_of_five << endl;
    // double
    cout << (double)square(5.3) << endl;
    // complex
    cout << square( complex<double>(5,3) )
         << endl;
    return 0; }
```

C++ does so far better job than **C** in this matter. But, with more **complex functions** the **syntax** can be hard **to follow**: see **this article** for example.

In **C++** you **must declare** that a **function** can work with **different types**. **Haskell** is so different in this matter. **Haskell function will be as general as possible by default**.

Type inference makes **Haskell** acts like **dynamically typed languages.** Unlike **dynamically typed languages,** most errors are caught before **run time.** Working with **Haskell** you can say:

if your code compiles, certainly it does what you intended

Type construction

You **can construct** your **own types**. First, you can use **aliases** or **type synonyms**.

```
--23tipos.hs

--type alias

type Nombre = String
type Color  = String
--function
showInfos :: Nombre ->  Color -> String
showInfos nom col =

  "Nombre: " ++ nom ++ ", Color: " ++ col

--values
nombre :: Name
nombre = "Diana"
color :: Color
color = "rosa"

--main program
main = putStrLn $ showInfos nombre color
```

```
Prelude> :l 23tipos                                         enter
[1 of 1] Compiling Main        ( 23tipos.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                 enter
Nombre: Diana, Color: rosa
```

But it doesn't protect you so much. Try **to swap showinfos function parameters** and run the **program**:

```
main = putStrLn $ showInfos color nombre
```

It will **compile** and **execute**. In fact you can replace **Name**, **Color** and **String** everywhere. The **compiler** will treat them as **completely identical.**

```
*Main> :r                                                   enter
[1 of 1] Compiling Main        ( 23tipos.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                 enter
Nombre: rosa, Color: Diana
```

Another **way** is to create your **own types** using **data keyword**.

```
--24tipos.hs
--datatype
data Nombre = NomConst String
data Color  = ColConst String
--function
showInfos :: Nombre ->  Color -> String
showInfos (NomConst nom) (ColConst col) =
  "Nombre: " ++ nom ++ ", Color: " ++ col
--values
nombre = NomConst "Giovanna"
color  = ColConst "azul"
--main program
main = putStrLn $ showInfos nombre color
```

```
Prelude> :l 24tipos                                             enter
[1 of 1] Compiling Main         ( 24tipos.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                     enter
Nombre: Giovanna, Color: azul
```

Now if you **swap showinfos function parameters**, the **compiler complains**! **So this is a potential mistake you will never make again and the only price is to be more verbose.**

Also notice that **data constructors** are **functions**:

```
NomConst :: String -> Nombre
ColConst :: String -> Color
```

The **data syntax** is mainly:

```
data TypeName = ConstructorName  [types]
              | ConstructorName2 [types]
              | ...
```

Most of the time we can use the same **name** for the **DataTypeName** and **DataConstructorName**.

Example:

```
data Complex a = Num a => Complex a a
```

Also you can use **record syntax**:

```
data DataTypeName = DataConstructor
  { field1 :: [type of field1]
  , field2 :: [type of field2]
  ...
  , fieldn :: [type of fieldn]
  }
```

And many **accessors (field1.. fieldn)** are made for you. Furthermore you can use another order when setting values. Example:

```
--25tiposRegistros.hs
{-# LANGUAGE DatatypeContexts #-}
--record syntax datatype
data Complex a = Num a => Complex
  { real :: a
  , img  :: a
  } deriving Show
--values
c = Complex 1.0 2.0
z = Complex {real = 3, img = 4}
--main program
main = putStrLn $  show c ++ " " ++ show z
```

```
Prelude> :l 25tiposRegistros                                    enter

tiposRegistros.hs:3:14: Warning:
    -XDatatypeContexts is deprecated: It was widely
considered a misfeature, and has been removed from the
Haskell language.
[1 of 1] Compiling Main ( 25tiposRegistros.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                     enter
Complex {real = 1.0, img = 2.0} Complex {real = 3, img = 4}
```

Recursive type

You already encountered a **recursive type**: **lists**. You can **re-create lists**, but with a more verbose syntax:

`data List a = Empty | Cons a (List a)`

If you really want to use an easier **syntax** you can use an **infix** name for **constructors**.

```
infixr 5 :::
data Lista a = Nil | a ::: (Lista a)
```

The number after **infixr** gives the **precedence**.

If you want to be able to **print (Show)**, **read (Read)**, test **equality (Eq)** and **compare (Ord)** your **new data structure** can tell **Haskell** to **derive** the appropriate **functions** for you.

When you add **deriving (Show)** to your **data declaration**, **Haskell** creates a **show function** for you. We'll see soon how you can use your **own show function**.

```
--26tiposRecursivos.hs
--infix constructor
infix 5 :::
--datatypes
data Lista a = Nil | a ::: (Lista a)
  deriving (Show, Read, Eq, Ord)

data List a = Empty | Cons a (List a)
  deriving (Show, Read, Eq, Ord)
--function
convierteLista []     = Nil
convierteLista (x:xs) = x ::: convierteLista xs
--main progam
main = do
  print (0 ::: (1 ::: Nil) )
  print (convierteLista [0,1])
  print (Cons 0 (Cons 1 Empty) )
```

```
Prelude> :l 26tiposRecursivos                                    enter
[1 of 1] Compiling Main (26tiposRecursivos.hs, interpreted)
Ok, modules loaded: Main.
*Main> main                                                      enter
0 ::: (1 ::: Nil)
0 ::: (1 ::: Nil)
Cons 0 (Cons 1 Empty)
```

Trees

We'll just give another standard example: **binary trees**. We will also create a **function** which **turns a list** into an **ordered binary tree**.

```
--27arboles.hs
import Data.List
--datatype
data BinTree a = Empty
                 | Node a (BinTree a) (BinTree a)
                 deriving (Show)
--function
treeFromList :: (Ord a) => [a] -> BinTree a
treeFromList []     = Empty
treeFromList (x:xs) = Node x (treeFromList . filter (<x) $ xs)
                             (treeFromList (filter (>x) xs))
--main program
main = do
  putStrLn “ingresa números seguidos por coma”
  lista <- getline
  print . treeFormList $ lista
```

```
Prelude> :l 27arboles                                            enter
[1 of 1] Compiling Main       ( 27arboles.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                      enter
```

```
ingresa numeros seguidos por coma
1,2,3
Node '1' (Node ',' Empty Empty) (Node '2' Empty (Node '3'
Empty Empty))
```
enter

Look at how elegant this function is. In plain English:

1. an **empty list** will **be converted** to an **empty tree**.
2. a **list (x:xs)** will **be converted** to a **tree** where:
   a. The **root is x**
   b. the **left subtree** is a **tree created** from **members** of the **list xs** which are strictly **inferior to x**
   c. the **right subtree** is a **tree created** from **members** of the **list xs** which are strictly **superior to x**.

This is an informative but quite unpleasant representation of our **tree**.

Just for fun, let's code a better display for our **trees**. I simply had fun making a nice **function** to **display trees** in a **general way**. You can safely skip this part if you find it too difficult to follow.

We have a few changes to make. We remove the **deriving (Show)** from the declaration of our **BinTree type**. And it might also be useful to make our **BinTree** an **instance of (Eq and Ord)** so we will be able to **test equality** and **compare trees**.

```
--28arboles.hs
-- datatype----------------------------------------
data BinTree a = Empty
                 | Node a (BinTree a) (BinTree a)
                 deriving (Eq, Ord)
-- instancia de BintTree a la clase de tipo Show-------------
instance (Show a) => Show (BinTree a) where
  show t = "< " ++ replace '\n' "\n: " (treeshow "" t)
    where
```

```
    treeshow pref Empty = ""
    treeshow pref (Node x Empty Empty) = (pshow pref x)

    treeshow pref (Node x left Empty) =
      (pshow pref x) ++ "\n" ++
        (showSon pref "*--" "   " left)

    treeshow pref (Node x Empty right) =
      (pshow pref x) ++ "\n" ++
        (showSon pref "*--" "   " right)

    treeshow pref (Node x left right) =
      (pshow pref x) ++ "\n" ++
        (showSon pref "|--" "|  " left) ++ "\n" ++
          (showSon pref "*--" "   " right)

    showSon pref before next t =
      pref ++ before ++ treeshow (pref ++ next) t

    pshow pref x = replace '\n' ("\n" ++ pref) (show x)

    replace c new string =
      concatMap (change c new) string
      where
        change c new x
            | x == c = new
            | otherwise = x:[]
-- functions----------------------------------------------------
--convierte una lista en árbol binario
```

```haskell
treeFromList :: Ord a => [a] -> BinTree a
treeFromList []     = Empty
treeFromList (x:xs) = Node x (treeFromList (filter (<x) xs) )
                             (treeFromList . filter (<x) $ xs)
--arbol infinito
nullTree :: BinTree Integer
nullTree = Node 0 nullTree nullTree

treeTakeDepth :: (Eq a1, Num a1) => a1 -> BinTree a -> BinTree a
treeTakeDepth _ Empty = Empty
treeTakeDepth 0 _     = Empty
treeTakeDepth n (Node x left right) =
  let
    nl = treeTakeDepth (n - 1) left
    nr = treeTakeDepth (n - 1) right
  in
  Node x nl nr

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap f Empty = Empty
treeMap f (Node x left right) = Node (f x)
  (treeMap f left)
  (treeMap f right)

infTreeTwo :: BinTree Int
infTreeTwo = Node 0 (treeMap (\x -> x - 1) infTreeTwo)
                    (treeMap (\x -> x + 1) infTreeTwo)
-- main program-------------------------------------------------
main = do
```

```
    putStrLn "Arbol binario de enteros:"
    print . treeFromList $ [7,2,4,8,1,3,6,21,12,23]
    putStrLn "************************************"
    putStrLn "Arbol binario de cadenas:"
    print . treeFromList $ ["foo","bar", "baz", "gor","yog"]
    putStrLn "************************************"
    putStrLn "Arbol binario de caracteres:"
    print . treeFromList . map treeFromList $ ["baz","zara","bar"]
    putStrLn "************************************"
    putStrLn "Arbol binario de 4 niveles con ceros"
    print . treeTakeDepth 4 $ nullTree
    putStrLn "************************************"
    putStrLn "Arbol con incrementos y decrementos"
    print . treeTakeDepth 4 $ infTreeTwo
```

```
Prelude> :l 28arboles                                         enter
[1 of 1] Compiling Main      ( 28arboles.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Arbol binario de enteros:                                     enter
< 7
: |--2
:    |--1
:    *--4
:       |--3
:       *--6
: *--8
:    *--21
:       |--12
:       *--23
************************************
Arbol binario de cadenas:
< "foo"
: |--"bar"
:    *--"baz"
: *--"gor"
:    *--"yog"
```

```
***********************************
Arbol binario de caracteres:
< < 'b'
: : |--'a'
: : *--'z'
: |--< 'b'
:    : |--'a'
:    : *--'r'
: *--< 'z'
:    : *--'a'
:    :   *--'r'
***********************************
Arbol binario de 4 niveles con ceros
< 0
: |--0
:    |--0
:       |--0
:       *--0
:    *--0
:       |--0
:       *--0
: *--0
:    |--0
:       |--0
:       *--0
:    *--0
:       |--0
:       *--0
***********************************
Arbol con incrementos y decrementos
< 0
: |---1
:    |---2
:       |---3
:       *---1
:    *--0
:       |---1
:       *--1
: *--1
:    |--0
:       |---1
:       *--1
:    *--2
```

```
:       |--1
:       *--3
```

Without **deriving Show**, **Haskell** doesn't create a **show function method** for us. Then we can create our **own version of show**. To achieve this, we must declare that our newly created **BinTree type** a is a **typeclass Show instance**. The **syntax** is:

```
instance Show (BinTree a) where
   show t = ... -- You declare your function here
```

Here is my own version to **show** a **binary tree**. Don't worry about the complexity. I made a lot of improvements in order to display even stranger objects. You can see in the **script** **arboles1.hs** how to declare **BinTree a** like a **Show typeclass instance.** Bellow box show you can write this **instance step by step**

```
-- declare BinTree a to be an instance of Show
instance (Show a) => Show (BinTree a) where
  -- will start by a '<' before the root
  -- and put a : a begining of line
  show t = "< " ++ replace '\n' "\n: " (treeshow "" t)
    where
    -- treeshow pref Tree
    -- shows a tree and starts each line with pref
    -- We don't display the Empty tree
    treeshow pref Empty = ""
    -- Leaf
    treeshow pref (Node x Empty Empty) =
                  (pshow pref x)

    -- Right branch is empty
    treeshow pref (Node x left Empty) =
                  (pshow pref x) ++ "\n" ++
                  (showSon pref "`--" "   " left)

    -- Left branch is empty
    treeshow pref (Node x Empty right) =
                  (pshow pref x) ++ "\n" ++
```

```
                 (showSon pref "`--" "   " right)

    -- Tree with left and right children non empty
    treeshow pref (Node x left right) =
                 (pshow pref x) ++ "\n" ++
                 (showSon pref "|--" "|  " left) ++ "\n" ++
                 (showSon pref "`--" "   " right)

    -- shows a tree using some prefixes to make it nice
    showSon pref before next t =
                 pref ++ before ++ treeshow (pref ++ next) t

    -- pshow replaces "\n" by "\n"++pref
    pshow pref x = replace '\n' ("\n"++pref) (show x)

    -- replaces one char by another string
    replace c new string =
      concatMap (change c new) string
      where
          change c new x
              | x == c = new
              | otherwise = x:[] -- "x"
```

Notice how **duplicated trees** aren't inserted; there is only one **tree** corresponding to **"I"**,**"HEARD"**. We have this for (almost) free, because we have declared **BinTree a** to be an **Eq instance**.

This structure is awesome: we can make **trees** containing not only **integers, strings and chars, but also other trees**. And we can even make a **tree containing a tree of trees**.