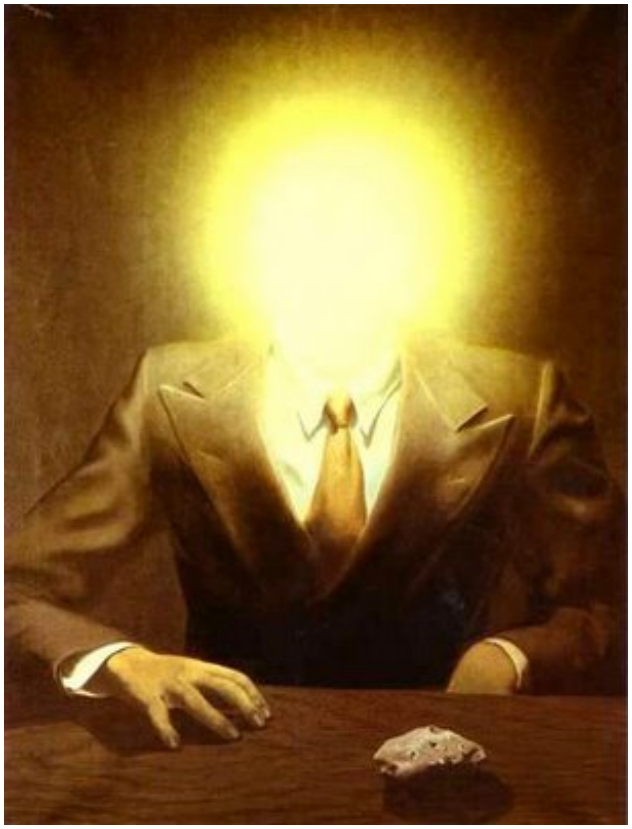
Learn Haskell Fast and Hard


yannesposito.com/Scratch/en/blog/Haskell-the-Hard-Way/



I really believe all developers should learn Haskell. I don't think everyone needs to be super Haskell ninjas, 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 article 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`
>
> *Some might not work, but most will. You should see a link just below.*

01_basic/10_Introduction/**00_hello_world.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 (quick sort, 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 `hello.hs` and:

```
~ runhaskell ./hello.hs
Hello World!
```

You could also download the literate Haskell source. You should see a link just above the introduction title. Download this file as `00_hello_world.lhs` and:

```
~ runhaskell 00_hello_world.lhs
Hello World!
```

01_basic/10_Introduction/**00_hello_world.lhs**

01_basic/10_Introduction/**10_hello_you.lhs**

Now, a program asking your name and replying "Hello" using the name you entered:

```
main = do
    print "What is your name?"
    name <- getLine
    print ("Hello " ++ name ++ "!")
```

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

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

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

```
#include <stdio.h>
int main (int argc, char **argv) {
    char name[666];

    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.

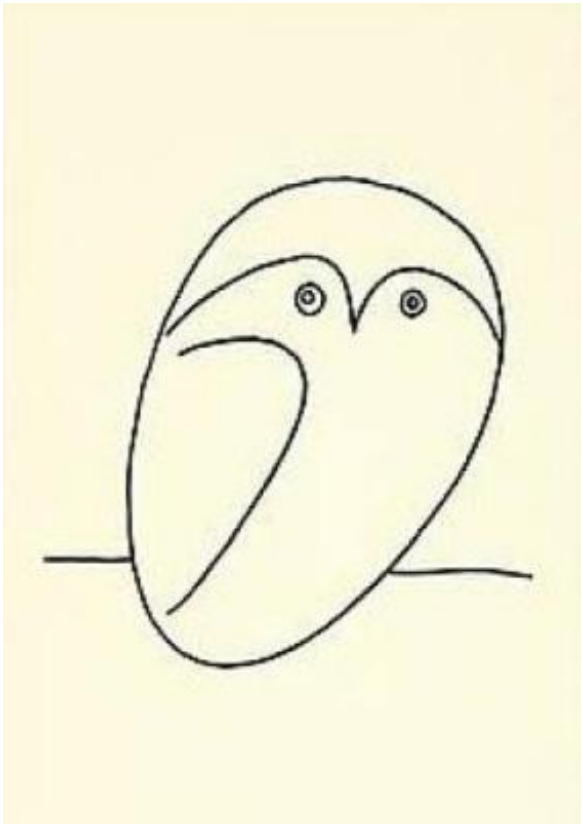
In Haskell there is a `main` function and every object has a type. The type of `main` is `IO ()`. This means `main` will cause side effects.

Just remember that Haskell can look a lot like mainstream imperative languages.

01_basic/10_Introduction/**10_hello_you.lhs**

01_basic/10_Introduction/**20_very_basic.lhs**

Very basic Haskell



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

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 in imperative languages.

Smart Static Typing

Instead of being in your way like in `C`, `C++` or `Java`, the type system is here to help you.

Purity

Generally your 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 with 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. For me, it is like reading scientific papers. Some parts are very clear, but when you see a formula, just focus and read slower. Also, while learning Haskell, it *really* doesn't matter much if you don't understand syntax details. If you meet a `>>=`, `<$>`, `<-` or any other weird symbol, just ignore them and follows the flow of the code.

Function declaration

You might be used to declaring functions like this:

In `C`:

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

In JavaScript:

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

in Python:

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

in Ruby:

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

In Scheme:

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

Finally, the 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 is not mandatory, type information for functions is usually made explicit. It's not mandatory because the compiler is smart enough to discover it for you. It's a good idea because it indicates intent and understanding.

Let's play a little. We declare the type using `::`

```
 f :: Int -> Int -> Int
 f x y = x*x + y*y

 main = print (f 2 3)
```

```
~ runhaskell 20_very_basic.lhs
13
```

01_basic/10_Introduction/**20_very_basic.lhs**

01_basic/10_Introduction/**21_very_basic.lhs**

Now try

```
f :: Int -> Int -> Int
f x y = x*x + y*y

main = print (f 2.3 4.2)
```

You should get this error:

```
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 `f', namely `4.2'
    In the first argument of `print', namely `(f 2.3 4.2)'
    In the expression: print (f 2.3 4.2)
```

The problem: `4.2` isn't an Int.

01_basic/10_Introduction/**21_very_basic.lhs**

01_basic/10_Introduction/**22_very_basic.lhs**

The solution: don't declare a type for `f` for the moment and let Haskell infer the most general type for us:

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

main = print (f 2.3 4.2)
```

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
GHCi, version 7.0.4: 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> let f x y = x*x + y*y
Prelude> :type f
f :: Num a => a -> a -> a
```

Uh? 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. `Num` contains only types which behave like numbers. More precisely, `Num` is class containing types which implement a specific list of functions, and in particular `(+)` and `(*)`.

Type classes are a very powerful language construct. We can do some incredibly powerful stuff with this. More on this later.

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

Let `a` be a type belonging to the `Num` type class. This is a function from type `a` to (`a -> a`).

Yes, strange. In fact, in Haskell no function really has two arguments. Instead all functions have only one argument. But we will note that taking two arguments is equivalent to taking 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.

01_basic/10_Introduction/**22_very_basic.lhs**

01_basic/10_Introduction/**23_very_basic.lhs**

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.

01_basic/10_Introduction/**23_very_basic.lhs**

01_basic/10_Introduction/**24_very_basic.lhs**

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

```
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)
```

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

01_basic/10_Introduction/**24_very_basic.lhs**

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 is a meta notation, ⇔ does not 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 different
```

Powers

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

`Integer` has no limit except the capacity of your machine:

```
4^103
10284403483257537763468557390983440656142099160209874145928806 4
```

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

```
$ ghci
....
Prelude> :m Data.Ratio
Data.Ratio> (11
11
```

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 . h) x   ⇔  f (g (h x))
```

01_basic/20_Essential_Haskell/**10a_Functions.lhs**

Useful notations for functions

Just a reminder:

```
x :: Int            ⇔ x is of 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 type class
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 type of a function 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!*

01_basic/20_Essential_Haskell/**10a_Functions.lhs**

Hard Part

The hard part can now begin.

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`

To show differences between functional and imperative approaches, I'll start by providing an imperative solution (in 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;
}
```

In Haskell, by contrast, we don't have variables or a for loop. One solution to achieve the same result without loops is to use recursion.

> *Remark: Recursion is generally perceived as slow in imperative languages. But this is generally not the case in functional programming. Most of the time Haskell will handle recursive functions efficiently.*

Here is a `C` version of the recursive function. 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) {
        return n;
    } else {
        x = list[0];
        xs = list+1;
        if ( 0 == (x%2) ) {
            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` verifies if a number is even.

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

`head` returns the first element of a list:

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

`tail` returns all elements of a list, except the first:

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

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

The first Haskell solution. The function `evenSum` returns the sum of all even numbers in a list:

```
evenSum :: [Integer] -> Integer

evenSum l = accumSum 0 l

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

To test a function you can use `ghci`:

```
% ghci
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load 11_Functions.lhs
[1 of 1] Compiling Main             ( 11_Functions.lhs, interpreted )
Ok, modules loaded: Main.
*Main> evenSum [1..5]
6
```

Here is an example of execution:

```
*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
```

02_Hard_Part/**11_Functions.lhs**

02_Hard_Part/**12_Functions.lhs**

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

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

evenSum l = accumSum 0 l
    where accumSum n l =
            if l == []
                then n
                else let x = head l
                         xs = tail l
                     in if even x
                           then accumSum (n+x) xs
                           else accumSum n xs
```

02_Hard_Part/**12_Functions.lhs**

02_Hard_Part/**13_Functions.lhs**

Next, we can use pattern matching.

```
evenSum l = accumSum 0 l
    where
        accumSum n [] = n
        accumSum n (x:xs) =
             if even x
                then accumSum (n+x) xs
                else accumSum n xs
```

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

Instead of saying: `foo l = if l == [] then <x> else <y>`You simply state:

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

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

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

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.

02_Hard_Part/**13_Functions.lhs**

02_Hard_Part/**14_Functions.lhs**

In Haskell you can simplify function definitions by η-reducing them. For example, instead of writing:

```
f x = (some expresion) x
```

you can simply write

```
f = some expression
```

We use this method to remove the `l`:

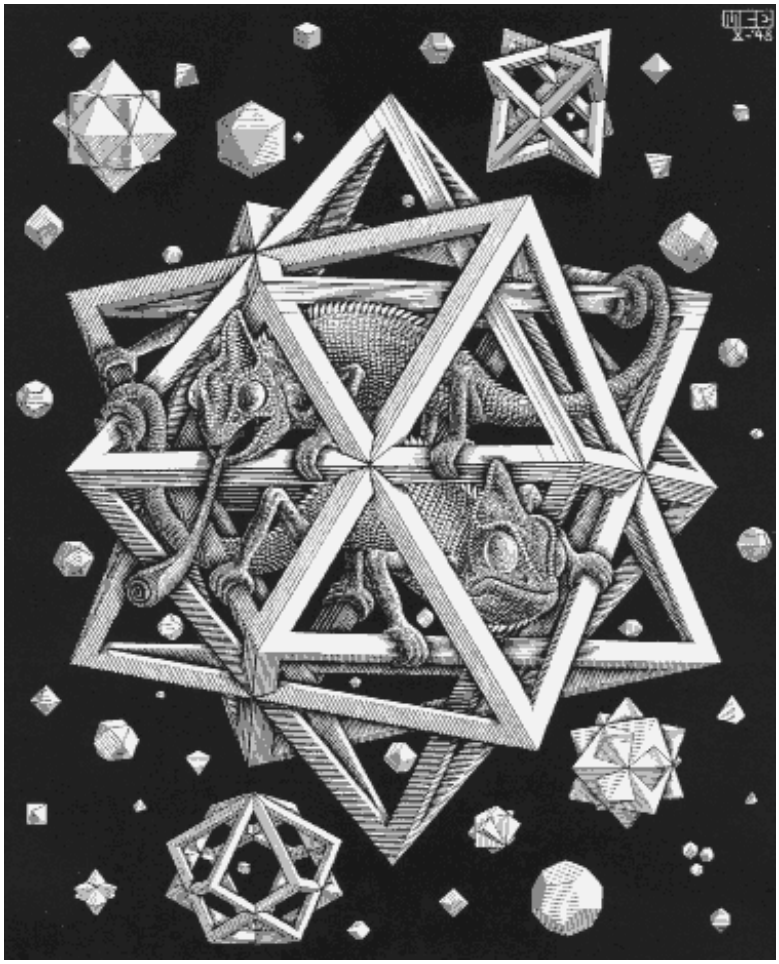
```
evenSum :: Integral a => [a] -> a

evenSum = accumSum 0
    where
        accumSum n [] = n
        accumSum n (x:xs) =
             if even x
                then accumSum (n+x) xs
                else accumSum n xs
```

02_Hard_Part/**14_Functions.lhs**

02_Hard_Part/**15_Functions.lhs**

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.

```
evenSum l = mysum 0 (filter even l)
    where
      mysum n [] = n
      mysum n (x:xs) = mysum (n+x) xs
```

where

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

The function `filter` takes a function of type (`a -> Bool`) and a list of type `[a]`. 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 function `foldl` 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 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 as Haskell is lazy, it doesn't evaluate `(f z x)` and simply pushes it onto the stack. This is why 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 identical.

Now our new version of `evenSum` becomes:

```
import Data.List
evenSum l = foldl' mysum 0 (filter even l)
  where mysum acc value = acc + value
```

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

```
import Data.List (foldl')
evenSum l = foldl' (\x y -> x+y) 0 (filter even l)
```

And of course, we note that

```
(\x y -> x+y) ⇔ (+)
```

02_Hard_Part/**15_Functions.lhs**

02_Hard_Part/**16_Functions.lhs**

Finally

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

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

To help you 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:

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

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

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

It is time to discuss the direction our code has moved as we introduced more functional idioms. What did we gain by using higher order functions?

At first, you might think the main difference is terseness. But in fact, it has more to do with better thinking. 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
```

Updating version 10 is extremely easy:

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

We just had to add another "transformation function"[^0216].

```
map (^2) [1,2,3,4] ⇔ [1,4,9,16]
```

The `map` function simply applies a function to all the elements of a list.

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 interchangably with others,

as needed. That is, you can compose, map, fold, filter using your new function.

Modifying version 1 is left as an exercise to the reader ☺.

If you believe we have reached the end of generalization, then know you are very 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.

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

02_Hard_Part/**16_Functions.lhs**

Types

> *tl;dr:*
>
> - `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 for you.*

In Haskell, types are strong and static.

Why is this important? It will help you *greatly* to avoid mistakes. In Haskell, most bugs are caught during the compilation of your program. And the main reason is because of the type inference during compilation. Type inference makes it easy to detect where you used the wrong parameter at the wrong place, for example.

Type inference

Static typing is generally essential for fast execution. But most statically typed languages are bad at generalizing concepts. Haskell's saving grace is that it can *infer* types.

Here is a simple example, the `square` function in Haskell:

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

This function can `square` any Numeral type. You can provide `square` with an `Int`, an `Integer`, a `Float` a `Fractional` and even `Complex`. Proof by example:

```
% ghci
GHCi, version 7.0.4:
...
Prelude> let square x = x*x
Prelude> square 2
4
Prelude> square 2.1
4.41
Prelude> -- load the Data.Complex module
Prelude> :m Data.Complex
Prelude Data.Complex> square (2 :+ 1)
3.0 :+ 4.0
```

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

Now compare with the amount of code necessary in C:

```
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 sqr_of_five = square(5);
    cout << sqr_of_five << endl;

    cout << (double)square(5.3) << endl;

    cout << square( complex<double>(5,3) )
         << endl;
    return 0;
}
```

C++ does a far better job than C in this regard. But for 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. In Haskell, the opposite is the case. The function will be as general as possible by default.

Type inference gives Haskell the feeling of freedom that dynamically typed languages provide. But unlike dynamically typed languages, most errors are caught before run time. Generally, in Haskell:

> *"if it compiles it certainly does what you intended"*

02_Hard_Part/**21_Types.lhs**

Type construction

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

```
type Name   = String
type Color  = String

showInfos :: Name ->  Color -> String
showInfos name color =  "Name: " ++ name
                        ++ ", Color: " ++ color
name :: Name
name = "Robin"
color :: Color
color = "Blue"
main = putStrLn $ showInfos name color
```

02_Hard_Part/**21_Types.lhs**

02_Hard_Part/**22_Types.lhs**

But it doesn't protect you much. Try to swap the two parameter of `showInfos` and run the program:

```
    putStrLn $ showInfos color name
```

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

Another method is to create your own types using the keyword `data`.

```
data Name   = NameConstr String
data Color  = ColorConstr String

showInfos :: Name ->  Color -> String
showInfos (NameConstr name) (ColorConstr color) =
      "Name: " ++ name ++ ", Color: " ++ color

name  = NameConstr "Robin"
color = ColorConstr "Blue"
main = putStrLn $ showInfos name color
```

Now if you switch parameters of `showInfos`, 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 constructors are functions:

```
NameConstr  :: String -> Name
ColorConstr :: String -> Color
```

The syntax of `data` is mainly:

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

Generally the usage is to use the same name for the DataTypeName and DataTypeConstructor.

Example:

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

Also you can use the record syntax:

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

And many accessors are made for you. Furthermore you can use another order when setting values.

Example:

```
data Complex a = Num a => Complex { real :: a, img :: a}
c = Complex 1.0 2.0
z = Complex { real = 3, img = 4 }
real c ⇒ 1.0
img z ⇒ 4
```

02_Hard_Part/**22_Types.lhs**

02_Hard_Part/**23_Types.lhs**

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 List a = Nil | a ::: (List 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 you can tell Haskell to derive the appropriate functions for you.

```
infixr 5 :::
data List a = Nil | a ::: (List a)
              deriving (Show,Read,Eq,Ord)
```

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.

```
convertList [] = Nil
convertList (x:xs) = x ::: convertList xs

main = do
      print (0 ::: 1 ::: Nil)
      print (convertList [0,1])
```

This prints:

```
0 ::: (1 ::: Nil)
0 ::: (1 ::: Nil)
```

02_Hard_Part/**23_Types.lhs**

02_Hard_Part/**30_Trees.lhs**

Trees

We'll just give another standard example: binary trees.

```
import Data.List

data BinTree a = Empty
                 | Node a (BinTree a)
(BinTree a)
                                 deriving (Show)
```

We will also create a function which turns a list into an ordered binary tree.

```
treeFromList :: (Ord a) => [a] -> BinTree a
treeFromList [] = Empty
treeFromList (x:xs) = Node x (treeFromList
(filter (<x) xs))
                               (treeFromList
(filter (>x) xs))
```

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

- an empty list will be converted to an empty tree.
- a list `(x:xs)` will be converted to a tree where:
- The root is `x`
- Its left subtree is the tree created from members of the list `xs` which are strictly inferior to `x` and
- the right subtree is the tree created from members of the list `xs` which are strictly superior to `x`.

```
main = print $ treeFromList [7,2,4,8]
```

You should obtain the following:

```
Node 7 (Node 2 Empty (Node 4 Empty Empty)) (Node 8 Empty Empty)
```

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

02_Hard_Part/**30_Trees.lhs**

02_Hard_Part/**31_Trees.lhs**

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.

```
data BinTree a = Empty
                 | Node a (BinTree a) (BinTree a)
                  deriving (Eq,Ord)
```

Without the `deriving (Show)`, Haskell doesn't create a `show` method for us. We will create our own version of

`show`. To achieve this, we must declare that our newly created type `BinTree a` is an instance of the type class `Show`. The general syntax is:

```
instance Show (BinTree a) where
   show t = ...
```

Here is my version of how to show a binary tree. Don't worry about the apparent complexity. I made a lot of improvements in order to display even stranger objects.

```
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:[]
```

The `treeFromList` method remains identical.

```
treeFromList :: (Ord a) => [a] -> BinTree a
treeFromList [] = Empty
treeFromList (x:xs) = Node x (treeFromList (filter (<x) xs))
                             (treeFromList (filter (>x) xs))
```

And now, we can play:

```
main = do
  putStrLn "Int binary tree:"
  print $ treeFromList [7,2,4,8,1,3,6,21,12,23]
```

```
Int binary tree:
< 7
: |
: |  |
: |  `
: |     |
: |     `
: `
:    `
:       |
:       `
```

Now it is far better! The root is shown by starting the line with the < character. And each following line starts with a :. But we could also use another type.

```
  putStrLn "\nString binary tree:"
  print $ treeFromList ["foo","bar","baz","gor","yog"]
```

```
String binary tree:
< "foo"
: |--"bar"
: |  `--"baz"
: `--"gor"
:    `--"yog"
```

As we can test equality and order trees, we can make tree of trees!

```
  putStrLn "\nBinary tree of Char binary trees:"
  print ( treeFromList
          (map treeFromList ["baz","zara","bar"]))
```

```
Binary tree of Char binary trees:
< < 'b'
: : |--'a'
: : `--'z'
: |--< 'b'
: |  : |--'a'
```

```
: |  : `--'r'
: `--< 'z'
:    : `--'a'
:    :    `--'r'
```

This is why I chose to prefix each line of tree display by `:` (except for the root).


```
  putStrLn "\nTree of Binary trees of Char binary trees:"
  print $ (treeFromList . map (treeFromList . map treeFromList))
            [ ["YO","DAWG"]
            , ["I","HEARD"]
            , ["I","HEARD"]
            , ["YOU","LIKE","TREES"] ]
```

Which is equivalent to

```
print ( treeFromList (
          map treeFromList
             [ map treeFromList ["YO","DAWG"]
             , map treeFromList ["I","HEARD"]
             , map treeFromList ["I","HEARD"]
             , map treeFromList ["YOU","LIKE","TREES"] ]))
```

and gives:

```
Binary tree of Binary trees of Char binary trees:
< < < 'Y'
: : : `--'O'
: : `--< 'D'
: :    : |--'A'
: :    : `--'W'
: :    :    `--'G'
: |--< < 'I'
: |  : `--< 'H'
: |  :    : |--'E'
: |  :    : |  `--'A'
```

```
: |  :    : |     `--'D'
: |  :    : `--'R'
: `--< < 'Y'
:    : : `--'O'
:    : :    `--'U'
:    : `--< 'L'
:    :    : `--'I'
:    :    :    |--'E'
:    :    :    `--'K'
:    :    `--< 'T'
:    :       : `--'R'
:    :       :    |--'E'
:    :       :    `--'S'
```

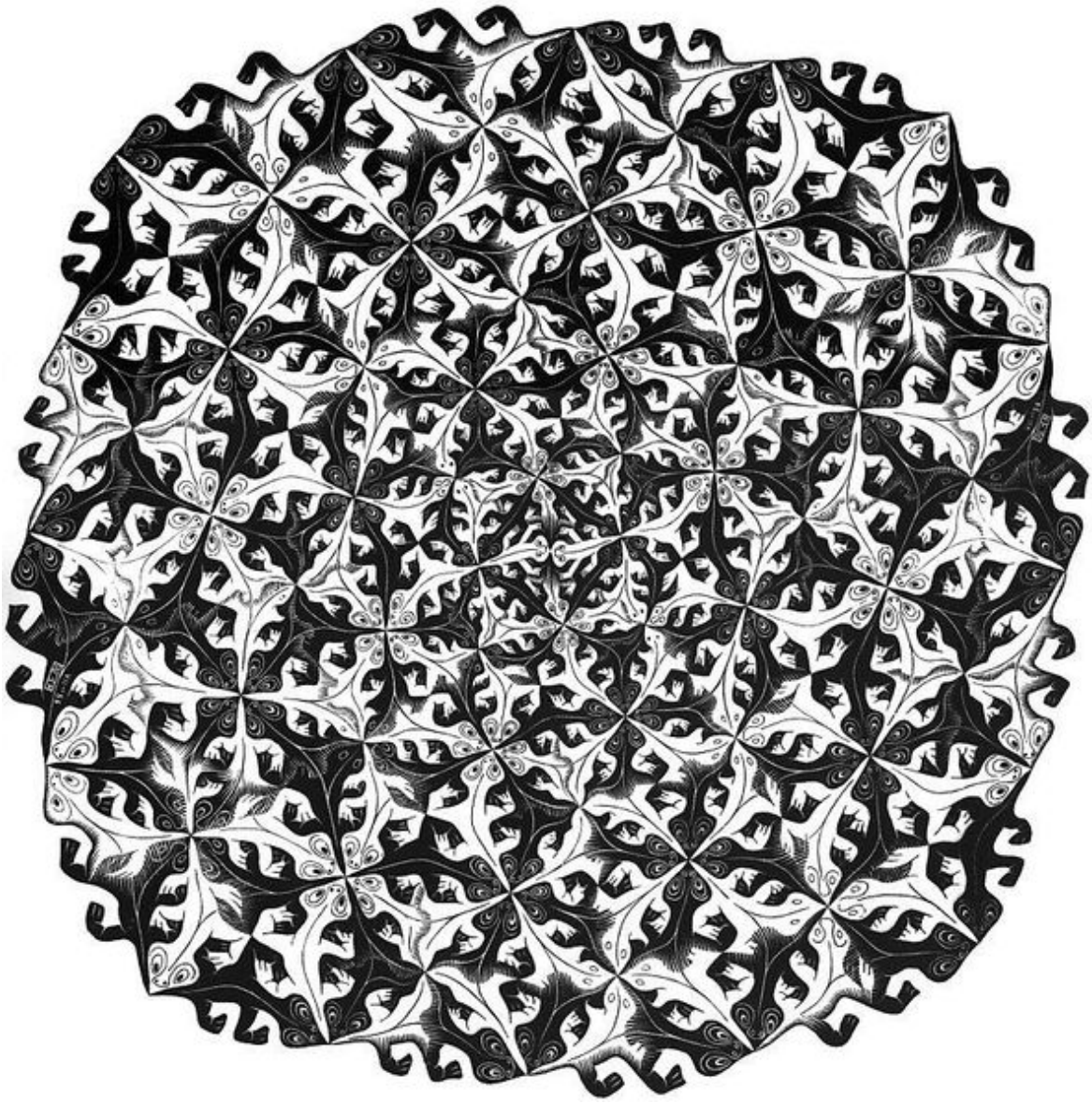
Notice how duplicate trees aren't inserted; there is only one tree corresponding to `"I","HEARD"`. We have this for (almost) free, because we have declared Tree to be an instance of `Eq`.

See how awesome this structure is: 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! fr:Voyez à quel point cette structure est formidable:

02_Hard_Part/**31_Trees.lhs**

02_Hard_Part/**40_Infinites_Structures.lhs**

Infinite Structures



It is often said that Haskell is *lazy*.

In fact, if you are a bit pedantic, you should say that Haskell is *non-strict*. Laziness is just a common implementation for non-strict languages.

Then what does “not-strict” mean? From the Haskell wiki:

> *Reduction (the mathematical term for evaluation) proceeds from the outside in.*
>
> *so if you have* `(a+(b*c))` *then you first reduce* `+` *first, then you reduce the inner* `(b*c)`

For example in Haskell you can do:

```
numbers :: [Integer]
numbers = 0:map (1+) numbers

take' n [] = []
```

```
take' 0 l = []
take' n (x:xs) = x:take' (n-1) xs

main = print $ take' 10 numbers
```

And it stops.

How?

Instead of trying to evaluate `numbers` entirely, it evaluates elements only when needed.

Also, note in Haskell there is a notation for infinite lists

```
[1..]   ⇔ [1,2,3,4...]
[1,3..] ⇔ [1,3,5,7,9,11...]
```

and most functions will work with them. Also, there is a built-in function `take` which is equivalent to our `take'`.

02_Hard_Part/**40_Infinites_Structures.lhs**

02_Hard_Part/**41_Infinites_Structures.lhs**

Suppose we don't mind having an ordered binary tree. Here is an infinite binary tree:

```
nullTree = Node 0 nullTree nullTree
```

A complete binary tree where each node is equal to 0. Now I will prove you can manipulate this object using the following function:

```
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
```

See what occurs for this program:

```
main = print $ treeTakeDepth 4 nullTree
```

This code compiles, runs and stops giving the following result:

```
<  0
: |
: |   |
: |   |   |
: |   |   `
: |   `
: |       |
: |       `
```

```
: `
:    |
:    |  |
:    |  `
:    `
:       |
:       `
```

Just to heat up your neurones a bit more, let's make a slightly more interesting tree:

```
iTree = Node 0 (dec iTree) (inc iTree)
        where
           dec (Node x l r) = Node (x-1) (dec l) (dec r)
           inc (Node x l r) = Node (x+1) (inc l) (inc r)
```

Another way to create this tree is to use a higher order function. This function should be similar to `map`, but should work on `BinTree` instead of list. Here is such a function:

```
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)
```

Hint: I won't talk more about this here. If you are interested in the generalization of `map` to other data structures, search for functor and `fmap`.

Our definition is now:

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

Look at the result for

```
main = print $ treeTakeDepth 4 infTreeTwo
```

```
<  0
: |
: |  |
: |  |  |
: |  |  `
: |  `
: |     |
: |     `
: `
:    |
:    |  |
:    |  `
:    `
:       |
```

: `

02_Hard_Part/**41_Infinites_Structures.lhs**

Hell Difficulty Part

Congratulations for getting so far! Now, some of the really hardcore stuff can start.

If you are like me, you should get the functional style. You should also understand a bit more the advantages of laziness by default. But you also don't really understand where to start in order to make a real program. And in particular:

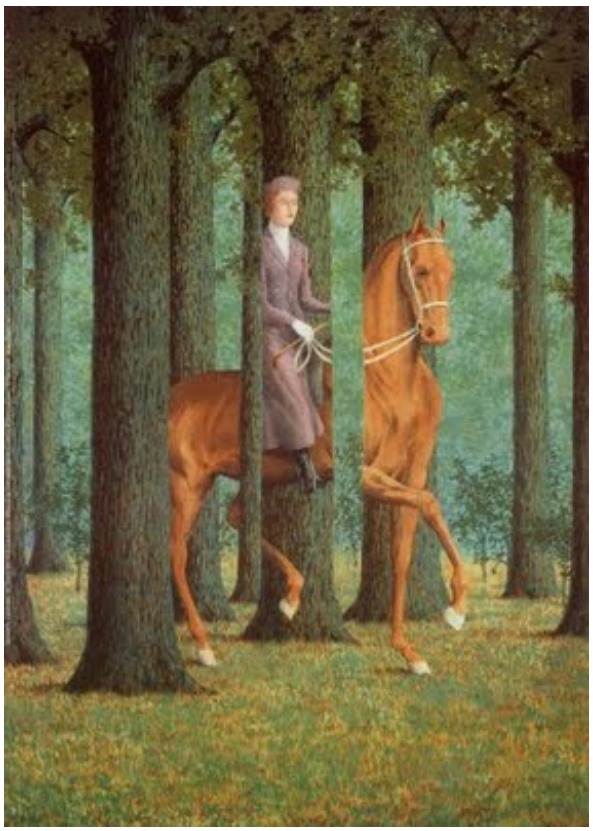
- How do you deal with effects?
- Why is there a strange imperative-like notation for dealing with IO?

Be prepared, the answers might be complex. But they are all very rewarding.

03_Hell/01_IO/**01_progressive_io_example.lhs**

Deal With IO

> *tl;dr:*
>
> *A typical function doing `IO` looks a lot like an imperative program:*
>
> ```
> f :: IO a
> f = do
>   x <- action1
>   action2 x
>   y <- action3
>   action4 x y
> ```
>
> - *To set a value to an object we use `<-`.*
> - *The type of each line is `IO *`; in this example:*
> - `action1 :: IO b`
> - `action2 x :: IO ()`
> - `action3 :: IO c`
> - `action4 x y :: IO a`
> - `x :: b, y :: c`
> - *Few objects have the type `IO a`, this should help you choose. In particular you cannot use pure functions directly here. To use pure functions you could do `action2 (purefunction x)` for example.*



In this section, I will explain how to use IO, not how it works. You'll see how Haskell separates the pure from the impure parts of the program.

Don’t stop because you’re trying to understand the details of the syntax. Answers will come in the next section.

What to achieve?

> *Ask a user to enter a list of numbers. Print the sum of the numbers*

```
toList :: String -> [Integer]
toList input = read ("[" ++ input ++ "]")

main = do
  putStrLn "Enter a list of numbers (separated by comma):"
  input <- getLine
  print $ sum (toList input)
```

It should be straightforward to understand the behavior of this program. Let’s analyze the types in more detail.

```
putStrLn :: String -> IO ()
getLine  :: IO String
print    :: Show a => a -> IO ()
```

Or more interestingly, we note that each expression in the `do` block has a type of `IO a`.

```
main = do
  putStrLn "Enter ... " :: IO ()
  getLine               :: IO String
  print Something       :: IO ()
```

We should also pay attention to the effect of the `<-` symbol.

```
do
 x <- something
```

If `something :: IO a` then `x :: a`.

Another important note about using `IO`: All lines in a do block must be of one of the two forms:

```
action1             :: IO a
                    -- in this case, generally a = ()
```

ou

```
value <- action2
```

These two kinds of line will correspond to two different ways of sequencing actions. The meaning of this sentence should be clearer by the end of the next section.

03_Hell/01_IO/**01_progressive_io_example.lhs**

03_Hell/01_IO/**02_progressive_io_example.lhs**

Now let's see how this program behaves. For example, what happens if the user enters something strange? Let's try:

```
% runghc 02_progressive_io_example.lhs
Enter a list of numbers (separated by comma):
foo
Prelude.read: no parse
```

Argh! An evil error message and a crash! Our first improvement will simply be to answer with a more friendly message.

In order to do this, we must detect that something went wrong. Here is one way to do this: use the type `Maybe`. This is a very common type in Haskell.

```
import Data.Maybe
```

What is this thing? `Maybe` is a type which takes one parameter. Its definition is:

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

This is a nice way to tell there was an error while trying to create/compute a value. The `maybeRead` function is a great example of this. This is a function similar to the function `read`, but if something goes wrong the returned value is `Nothing`. If the value is right, it returns `Just <the value>`. Don't try to understand too much of this function. I use a lower level function than `read`; `reads`.

```
maybeRead :: Read a => String -> Maybe a
maybeRead s = case reads s of
                  [(x,"")]    -> Just x
                  _           -> Nothing
```

Now to be a bit more readable, we define a function which goes like this: If the string has the wrong format, it will return `Nothing`. Otherwise, for example for "1,2,3", it will return `Just [1,2,3]`.

```
getListFromString :: String -> Maybe [Integer]
getListFromString str = maybeRead $ "[" ++ str ++ "]"
```

We simply have to test the value in our main function.

```
main :: IO ()
main = do
  putStrLn "Enter a list of numbers (separated by comma):"
  input <- getLine
  let maybeList = getListFromString input in
      case maybeList of
          Just l  -> print (sum l)
          Nothing -> error "Bad format. Good Bye."
```

In case of error, we display a nice error message.

Note that the type of each expression in the main's `do` block remains of the form `IO a`. The only strange construction is `error`. I'll just say here that `error msg` takes the needed type (here `IO ()`).

One very important thing to note is the type of all the functions defined so far. There is only one function which

contains `IO` in its type: `main`. This means main is impure. But main uses `getListFromString` which is pure. So it's clear just by looking at declared types which functions are pure and which are impure.

Why does purity matter? Among the many advantages, here are three:

- It is far easier to think about pure code than impure code.
- Purity protects you from all the hard-to-reproduce bugs that are due to side effects.
- You can evaluate pure functions in any order or in parallel without risk.

This is why you should generally put as most code as possible inside pure functions.

03_Hell/01_IO/**02_progressive_io_example.lhs**

03_Hell/01_IO/**03_progressive_io_example.lhs**

Our next iteration will be to prompt the user again and again until she enters a valid answer.

We keep the first part:

```
import Data.Maybe

maybeRead :: Read a => String -> Maybe a
maybeRead s = case reads s of
                  [(x,"")]    -> Just x
                  _           -> Nothing
getListFromString :: String -> Maybe [Integer]
getListFromString str = maybeRead $ "[" ++ str ++ "]"
```

Now we create a function which will ask the user for an list of integers until the input is right.

```
askUser :: IO [Integer]
askUser = do
  putStrLn "Enter a list of numbers (separated by comma):"
  input <- getLine
  let maybeList = getListFromString input in
      case maybeList of
          Just l  -> return l
          Nothing -> askUser
```

This function is of type `IO [Integer]`. Such a type means that we retrieved a value of type `[Integer]` through some IO actions. Some people might explain while waving their hands:

> *«This is an `[Integer]` inside an `IO`»*

If you want to understand the details behind all of this, you'll have to read the next section. But really, if you just want to *use* IO just practice a little and remember to think about the type.

Finally our main function is much simpler:

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

```
main = do
  list <- askUser
  print $ sum list
```

We have finished with our introduction to `IO`. This was quite fast. Here are the main things to remember:

- in the `do` block, each expression must have the type `IO a`. You are then limited in the number of expressions available. For example, `getLine`, `print`, `putStrLn`, etc…
- Try to externalize the pure functions as much as possible.
- the `IO a` type means: an IO *action* which returns an element of type `a`. `IO` represents actions; under the hood, `IO a` is the type of a function. Read the next section if you are curious.

If you practice a bit, you should be able to *use* `IO`.

> *Exercises:*
>
> - *Make a program that sums all of its arguments. Hint: use the function `getArgs`.*

03_Hell/01_IO/**03_progressive_io_example.lhs**

IO trick explained

> *Here is a tl;dr: for this section.*
>
> *To separate pure and impure parts, `main` is defined as a function which modifies the state of the world*
>
> *`main :: World -> World`*

> *A function is guaranteed to have side effects only if it has this type. But look at a typical main function:*
>
> ```
> main w0 =
>     let (v1,w1) = action1 w0 in
>     let (v2,w2) = action2 v1 w1 in
>     let (v3,w3) = action3 v2 w2 in
>     action4 v3 w3
> ```
>
> *We have a lot of temporary elements (here `w1`, `w2` and `w3`) which must be passed on to the next action.*
>
> *We create a function `bind` or `(>>=)`. With `bind` we don't need temporary names anymore.*
>
> ```
> main =
>   action1 >>= action2 >>= action3 >>= action4
> ```
>
> *Bonus: Haskell has syntactical sugar for us:*
>
> ```
> main = do
>   v1 <- action1
>   v2 <- action2 v1
>   v3 <- action3 v2
>   action4 v3
> ```

Why did we use this strange syntax, and what exactly is this `IO` type? It looks a bit like magic.

For now let's just forget all about the pure parts of our program, and focus on the impure parts:

```
askUser :: IO [Integer]
askUser = do
  putStrLn "Enter a list of numbers (separated by commas):"
  input <- getLine
  let maybeList = getListFromString input in
      case maybeList of
          Just l  -> return l
          Nothing -> askUser

main :: IO ()
main = do
  list <- askUser
  print $ sum list
```

First remark: this looks imperative. Haskell is powerful enough to make impure code look imperative. For example, if you wish you could create a `while` in Haskell. In fact, for dealing with `IO`, an imperative style is generally more appropriate.

But you should have noticed that the notation is a bit unusual. Here is why, in detail.

In an impure language, the state of the world can be seen as a huge hidden global variable. This hidden variable is accessible by all functions of your language. For example, you can read and write a file in any function. Whether a file exists or not is a difference in the possible states that the world can take.

In Haskell this state is not hidden. Rather, it is *explicitly* said that `main` is a function that *potentially* changes the state of the world. Its type is then something like:

```
main :: World -> World
```

Not all functions may have access to this variable. Those which have access to this variable are impure. Functions to which the world variable isn't provided are pure.

Haskell considers the state of the world as an input variable to `main`. But the real type of main is closer to this one:

```
main :: World -> ((),World)
```

The `()` type is the unit type. Nothing to see here.

Now let's rewrite our main function with this in mind:

```
main w0 =
    let (list,w1) = askUser w0 in
    let (x,w2) = print (sum list,w1) in
    x
```

First, we note that all functions which have side effects must have the type:

```
World -> (a,World)
```

where `a` is the type of the result. For example, a `getChar` function should have the type `World -> (Char,World)`.

Another thing to note is the trick to fix the order of evaluation. In Haskell, in order to evaluate `f a b`, you have many choices:

- first eval `a` then `b` then `f a b`
- first eval `b` then `a` then `f a b`.
- eval `a` and `b` in parallel then `f a b`

This is true because we're working in a pure part of the language.

Now, if you look at the main function, it is clear you must eval the first line before the second one since to evaluate the second line you have to get a parameter given by the evaluation of the first line.

This trick works nicely. The compiler will at each step provide a pointer to a new real world id. Under the hood, `print` will evaluate as:

- print something on the screen
- modify the id of the world
- evaluate as `((),new world id)`.

Now, if you look at the style of the main function, it is clearly awkward. Let's try to do the same to the askUser function:

```
askUser :: World -> ([Integer],World)
```

Before:

```
askUser :: IO [Integer]
askUser = do
  putStrLn "Enter a list of numbers:"
  input <- getLine
  let maybeList = getListFromString input in
      case maybeList of
          Just l  -> return l
          Nothing -> askUser
```

After:

```
askUser w0 =
    let (_,w1)     = putStrLn "Enter a list of numbers:" in
    let (input,w2) = getLine w1 in
    let (l,w3)     = case getListFromString input of
                       Just l   -> (l,w2)
                       Nothing  -> askUser w2
    in
        (l,w3)
```

This is similar, but awkward. Look at all these temporary `w?` names.

The lesson is: naive IO implementation in Pure functional languages is awkward!

Fortunately, there is a better way to handle this problem. We see a pattern. Each line is of the form:

```
let (y,w') = action x w in
```

Even if for some line the first `x` argument isn't needed. The output type is a couple, `(answer, newWorldValue)`. Each function `f` must have a type similar to:

```
f :: World -> (a,World)
```

Not only this, but we can also note that we always follow the same usage pattern:

```
let (y,w1) = action1 w0 in
let (z,w2) = action2 w1 in
let (t,w3) = action3 w2 in
...
```

Each action can take from 0 to n parameters. And in particular, each action can take a parameter from the result of a line above.

For example, we could also have:

```
let (_,w1) = action1 x w0   in
let (z,w2) = action2 w1     in
let (_,w3) = action3 x z w2 in
...
```

And of course `actionN w :: (World) -> (a,World)`.

> *IMPORTANT: there are only two important patterns to consider:*

> *let (x,w1) = action1 w0 in*
> *let (y,w2) = action2 x w1 in*
>
> *and*
>
> *let (_,w1) = action1 w0 in*
> *let (y,w2) = action2 w1 in*

Now, we will do a magic trick. We will make the temporary world symbol "disappear". We will `bind` the two lines. Let's define the `bind` function. Its type is quite intimidating at first:

```
bind :: (World -> (a,World))
        -> (a -> (World -> (b,World)))
        -> (World -> (b,World))
```

But remember that `(World -> (a,World))` is the type for an IO action. Now let's rename it for clarity:

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

Some examples of functions:

```
getLine :: IO String
print :: Show a => a -> IO ()
```

`getLine` is an IO action which takes world as a parameter and returns a couple `(String,World)`. This can be summarized as: `getLine` is of type `IO String`, which we also see as an IO action which will return a String "embeded inside an IO".

The function `print` is also interesting. It takes one argument which can be shown. In fact it takes two arguments. The first is the value to print and the other is the state of world. It then returns a couple of type `((),World)`. This means that it changes the state of the world, but doesn't yield any more data.

This type helps us simplify the type of `bind`:

```
bind :: IO a
        -> (a -> IO b)
        -> IO b
```

It says that `bind` takes two IO actions as parameters and returns another IO action.

Now, remember the *important* patterns. The first was:

```
let (x,w1) = action1 w0 in
let (y,w2) = action2 x w1 in
(y,w2)
```

Look at the types:

```
action1  :: IO a
action2  :: a -> IO b
(y,w2)   :: IO b
```

Doesn't it seem familiar?

```
(bind action1 action2) w0 =
    let (x, w1) = action1 w0
        (y, w2) = action2 x w1
    in  (y, w2)
```

The idea is to hide the World argument with this function. Let's go: As an example imagine if we wanted to simulate:

```
let (line1,w1) = getLine w0 in
let ((),w2) = print line1 in
((),w2)
```

Now, using the bind function:

```
(res,w2) = (bind getLine (\l -> print l)) w0
```

As print is of type `(World -> ((),World))`, we know `res = ()` (null type). If you didn't see what was magic here, let's try with three lines this time.

```
let (line1,w1) = getLine w0 in
let (line2,w2) = getLine w1 in
let ((),w3) = print (line1 ++ line2) in
((),w3)
```

Which is equivalent to:

```
(res,w3) = (bind getLine (\line1 ->
             (bind getLine (\line2 ->
               print (line1 ++ line2))))) w0
```

Didn't you notice something? Yes, no temporary World variables are used anywhere! This is *MA*. *GIC*.

We can use a better notation. Let's use `(>>=)` instead of `bind`. `(>>=)` is an infix function like `(+)`; reminder `3 + 4 ⇔ (+) 3 4`

```
(res,w3) = (getLine >>=
           (\line1 -> getLine >>=
           (\line2 -> print (line1 ++ line2)))) w0
```

fr; HAskell a un sucre syntaxique pour nous: Ho Ho Ho! Merry Christmas Everyone! Haskell has made syntactical sugar for us:

```
do
  x <- action1
  y <- action2
  z <- action3
  ...
```

Is replaced by:

```
action1 >>= (\x ->
action2 >>= (\y ->
action3 >>= (\z ->
```

```
...
)))
```

Note that you can use `x` in `action2` and `x` and `y` in `action3`.

But what about the lines not using the `<-`? Easy, another function `blindBind`:

```
blindBind :: IO a -> IO b -> IO b
blindBind action1 action2 w0 =
    bind action (\_ -> action2) w0
```

I didn't simplify this definition for the purposes of clarity. Of course we can use a better notation, we'll use the `(>>)` operator.

And

```
do
    action1
    action2
    action3
```

Is transformed into

```
action1 >>
action2 >>
action3
```

Also, another function is quite useful.

```
putInIO :: a -> IO a
putInIO x = IO (\w -> (x,w))
```

This is the general way to put pure values inside the "IO context". The general name for `putInIO` is `return`. This is quite a bad name when you learn Haskell. `return` is very different from what you might be used to.

03_Hell/01_IO/**21_Detailled_IO.lhs**

To finish, let's translate our example:

```
askUser :: IO [Integer]
askUser = do
  putStrLn "Enter a list of numbers (separated by commas):"
  input <- getLine
  let maybeList = getListFromString input in
      case maybeList of
          Just l  -> return l
          Nothing -> askUser

main :: IO ()
main = do
  list <- askUser
  print $ sum list
```

Is translated into:

```
import Data.Maybe

maybeRead :: Read a => String -> Maybe a
maybeRead s = case reads s of
                  [(x,"")]    -> Just x
                  _           -> Nothing
getListFromString :: String -> Maybe [Integer]
getListFromString str = maybeRead $ "[" ++ str ++ "]"
askUser :: IO [Integer]
askUser =
    putStrLn "Enter a list of numbers (sep. by commas):" >>
    getLine >>= \input ->
    let maybeList = getListFromString input in
      case maybeList of
        Just l -> return l
        Nothing -> askUser

main :: IO ()
main = askUser >>=
  \list -> print $ sum list
```

You can compile this code to verify that it works.

Imagine what it would look like without the `(>>)` and `(>>=)`.

03_Hell/01_IO/**21_Detailled_IO.lhs**

03_Hell/02_Monads/**10_Monads.lhs**

Monads

Now the secret can be revealed: `IO` is a *monad*. Being a monad means you have access to some syntactical sugar with the `do` notation. But mainly, you have access to a coding pattern which will ease the flow of your code.

> ***Important remarks**:*
>
> - *Monad are not necessarily about effects! There are a lot of pure monads.*
> - *Monad are more about sequencing*

In Haskell, `Monad` is a type class. To be an instance of this type class, you must provide the functions `(>>=)` and `return`. The function `(>>)` is derived from `(>>=)`. Here is how the type class `Monad` is declared (basically):

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

  (>>) :: m a -> m b -> m b
  f >> g = f >>= \_ -> g



  fail :: String -> m a
  fail = error
```

> *Remarks:*
>
> - *the keyword `class` is not your friend. A Haskell class is not a class of the kind you will find in object-oriented programming. A Haskell class has a lot of similarities with Java interfaces. A better word would have been `typeclass`, since that means a set of types. For a type to belong to a class, all functions of the class must be provided for this type.*
> - *In this particular example of type class, the type `m` must be a type that takes an argument. for example `IO a`, but also `Maybe a`, `[a]`, etc…*
> - *To be a useful monad, your function must obey some rules. If your construction does not obey these rules strange things might happens:*
>
> *~ return a >>= k == k a m >>= return == m m >>= (-> k x >>= h) == (m >>= k) >>= h ~*

Maybe is a monad

There are a lot of different types that are instances of `Monad`. One of the easiest to describe is `Maybe`. If you have a sequence of `Maybe` values, you can use monads to manipulate them. It is particularly useful to remove very deep `if..then..else..` constructions.

Imagine a complex bank operation. You are eligible to gain about 700€ only if you can afford to follow a list of operations without your balance dipping below zero.

```
deposit  value account = account + value
```

```
withdraw value account = account - value

eligible :: (Num a,Ord a) => a -> Bool
eligible account =
  let account1 = deposit 100 account in
    if (account1 < 0)
    then False
    else
      let account2 = withdraw 200 account1 in
      if (account2 < 0)
      then False
      else
        let account3 = deposit 100 account2 in
        if (account3 < 0)
        then False
        else
          let account4 = withdraw 300 account3 in
          if (account4 < 0)
          then False
          else
            let account5 = deposit 1000 account4 in
            if (account5 < 0)
            then False
            else
              True

main = do
  print $ eligible 300
  print $ eligible 299
```

03_Hell/02_Monads/**10_Monads.lhs**

03_Hell/02_Monads/**11_Monads.lhs**

Now, let's make it better using Maybe and the fact that it is a Monad

```
deposit :: (Num a) => a -> a -> Maybe a
deposit value account = Just (account + value)

withdraw :: (Num a,Ord a) => a -> a -> Maybe a
withdraw value account = if (account < value)
                         then Nothing
                         else Just (account - value)

eligible :: (Num a, Ord a) => a -> Maybe Bool
eligible account = do
  account1 <- deposit 100 account
  account2 <- withdraw 200 account1
  account3 <- deposit 100 account2
  account4 <- withdraw 300 account3
```

```
  account5 <- deposit 1000 account4
  Just True

main = do
  print $ eligible 300
  print $ eligible 299
```

03_Hell/02_Monads/**11_Monads.lhs**

03_Hell/02_Monads/**12_Monads.lhs**

Not bad, but we can make it even better:

```
deposit :: (Num a) => a -> a -> Maybe a
deposit value account = Just (account + value)

withdraw :: (Num a,Ord a) => a -> a -> Maybe a
withdraw value account = if (account < value)
                         then Nothing
                         else Just (account - value)

eligible :: (Num a, Ord a) => a -> Maybe Bool
eligible account =
  deposit 100 account >>=
  withdraw 200 >>=
  deposit 100  >>=
  withdraw 300 >>=
  deposit 1000 >>
  return True

main = do
  print $ eligible 300
  print $ eligible 299
```

We have proven that Monads are a good way to make our code more elegant. Note this idea of code organization, in particular for `Maybe` can be used in most imperative languages. In fact, this is the kind of construction we make naturally.

> *An important remark:*
>
> *The first element in the sequence being evaluated to `Nothing` will stop the complete evaluation. This means you don't execute all lines. You get this for free, thanks to laziness.*

You could also replay these example with the definition of `(>>=)` for `Maybe` in mind:

```
instance Monad Maybe where
    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing  >>= _   = Nothing
    (Just x) >>= f   = f x
```

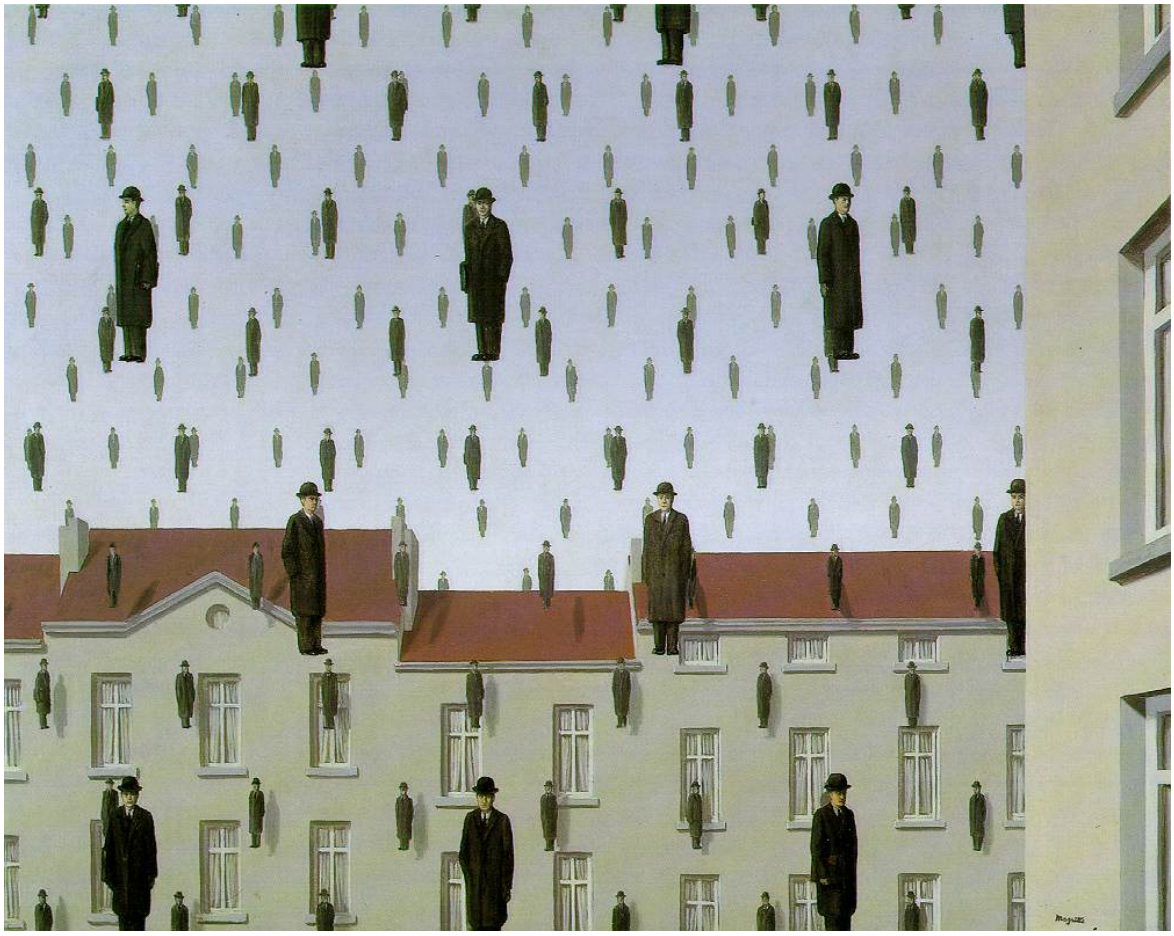
```
    return x = Just x
```

The `Maybe` monad proved to be useful while being a very simple example. We saw the utility of the `IO` monad. But now for a cooler example, lists.

03_Hell/02_Monads/**12_Monads.lhs**

03_Hell/02_Monads/**13_Monads.lhs**

The list monad



The list monad helps us to simulate non-deterministic computations. Here we go:

```
import Control.Monad (guard)

allCases = [1..10]

resolve :: [(Int,Int,Int)]
resolve = do
              x <- allCases
              y <- allCases
              z <- allCases
              guard $ 4*x + 2*y < z
              return (x,y,z)
```

```
main = do
  print resolve
```

MA. GIC. :

```
[(1,1,7),(1,1,8),(1,1,9),(1,1,10),(1,2,9),(1,2,10)]
```

For the list monad, there is also this syntactic sugar:

```
  print $ [ (x,y,z) | x <- allCases,
                      y <- allCases,
                      z <- allCases,
                      4*x + 2*y < z ]
```

I won't list all the monads, but there are many of them. Using monads simplifies the manipulation of several notions in pure languages. In particular, monads are very useful for:

- IO,
- non-deterministic computation,
- generating pseudo random numbers,
- keeping configuration state,
- writing state,
- …

If you have followed me until here, then you've done it! You know monads!

03_Hell/02_Monads/**13_Monads.lhs**

Appendix

This section is not so much about learning Haskell. It is just here to discuss some details further.

04_Appendice/01_More_on_infinite_trees/**10_Infinite_Trees.lhs**

More on Infinite Tree

In the section Infinite Structures we saw some simple constructions. Unfortunately we removed two properties from our tree:

1. no duplicate node value
2. well ordered tree

In this section we will try to keep the first property. Concerning the second one, we must relax it but we'll discuss how to keep it as much as possible.

Our first step is to create some pseudo-random number list:

```
shuffle = map (\x -> (x*3123) `mod` 4331) [1..]
```

Just as a reminder, here is the definition of `treeFromList`

```
treeFromList :: (Ord a) => [a] -> BinTree a
treeFromList []    = Empty
treeFromList (x:xs) = Node x (treeFromList (filter (<x) xs))
                             (treeFromList (filter (>x) xs))
```

and `treeTakeDepth`:

```
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
```

See the result of:

```
main = do
      putStrLn "take 10 shuffle"
      print $ take 10 shuffle
      putStrLn "\ntreeTakeDepth 4 (treeFromList shuffle)"
      print $ treeTakeDepth 4 (treeFromList shuffle)
```

```
% runghc 02_Hard_Part/41_Infinites_Structures.lhs
take 10 shuffle
[3123,1915,707,3830,2622,1414,206,3329,2121,913]
treeTakeDepth 4 (treeFromList shuffle)

< 3123
: |
: |  |
: |  |  |
: |  |  `
: |  `
: |     |
: |     `
: `
:    |
:    |  |
:    |  `
:    `
:       |
:       `
```

Yay! It ends! Beware though, it will only work if you always have something to put into a branch.

For example

```
treeTakeDepth 4 (treeFromList [1..])
```

will loop forever. Simply because it will try to access the head of `filter (<1) [2..]`. But `filter` is not smart enought to understand that the result is the empty list.

Nonetheless, it is still a very cool example of what non strict programs have to offer.

Left as an exercise to the reader:

- Prove the existence of a number `n` so that `treeTakeDepth n (treeFromList shuffle)` will enter an infinite loop.
- Find an upper bound for `n`.
- Prove there is no `shuffle` list so that, for any depth, the program ends.

04_Appendice/01_More_on_infinite_trees/**10_Infinite_Trees.lhs**

04_Appendice/01_More_on_infinite_trees/**11_Infinite_Trees.lhs**

In order to resolve these problem we will modify slightly our `treeFromList` and `shuffle` function.

A first problem, is the lack of infinite different number in our implementation of `shuffle`. We generated only `4331` different numbers. To resolve this we make a slightly better `shuffle` function.

```
shuffle = map rand [1..]
          where
              rand x = ((p x) `mod` (x+c)) - ((x+c) `div` 2)
              p x = m*x^2 + n*x + o
              m = 3123
              n = 31
              o = 7641
              c = 1237
```

This shuffle function has the property (hopefully) not to have an upper nor lower bound. But having a better shuffle list isn't enough not to enter an infinite loop.

Generally, we cannot decide whether `filter (<x) xs` is empty. Then to resolve this problem, I'll authorize some error in the creation of our binary tree. This new version of code can create binary tree which don't have the following property for some of its nodes:

> *Any element of the left (resp. right) branch must all be strictly inferior (resp. superior) to the label of the root.*

Remark it will remains *mostly* an ordered binary tree. Furthermore, by construction, each node value is unique in the tree.

Here is our new version of `treeFromList`. We simply have replaced `filter` by `safefilter`.

```
treeFromList :: (Ord a, Show a) => [a] -> BinTree a
treeFromList []     = Empty
treeFromList (x:xs) = Node x left right
          where
              left = treeFromList $ safefilter (<x) xs
              right = treeFromList $ safefilter (>x) xs
```

This new function `safefilter` is almost equivalent to `filter` but don't enter infinite loop if the result is a finite list. If it cannot find an element for which the test is true after 10000 consecutive steps, then it considers to be the end of the search.

```
safefilter :: (a -> Bool) -> [a] -> [a]
safefilter f l = safefilter' f l nbTry
  where
      nbTry = 10000
      safefilter' _ _ 0 = []
      safefilter' _ [] _ = []
      safefilter' f (x:xs) n =
                  if f x
                     then x : safefilter' f xs nbTry
                     else safefilter' f xs (n-1)
```

Now run the program and be happy:

```
main = do
      putStrLn "take 10 shuffle"
      print $ take 10 shuffle
      putStrLn "\ntreeTakeDepth 8 (treeFromList shuffle)"
      print $ treeTakeDepth 8 (treeFromList $ shuffle)
```

You should realize the time to print each value is different. This is because Haskell compute each value when it needs it. And in this case, this is when asked to print it on the screen.

Impressively enough, try to replace the depth from `8` to `100`. It will work without killing your RAM! The flow and the memory management is done naturally by Haskell.

Left as an exercise to the reader:

- Even with large constant value for `deep` and `nbTry`, it seems to work nicely. But in the worst case, it can be exponential. Create a worst case list to give as parameter to `treeFromList`.
  hint: think about (`[0,-1,-1,....,-1,1,-1,...,-1,1,...]`).
- I first tried to implement `safefilter` as follow:

  ```
  safefilter' f l = if filter f (take 10000 l) == []
                    then []
                    else filter f l
  ```

  Explain why it doesn't work and can enter into an infinite loop.
- Suppose that `shuffle` is real random list with growing bounds. If you study a bit this structure, you'll discover that with probability 1, this structure is finite. Using the following code (suppose we could use `safefilter'` directly as if was not in the where of safefilter) find a definition of `f` such that with probability 1, `treeFromList' shuffle` is infinite. And prove it. Disclaimer, this is only a conjecture.

```
treeFromList' []  n = Empty
treeFromList' (x:xs) n = Node x left right
    where
        left = treeFromList' (safefilter' (<x) xs (f n)
```

```
        right = treeFromList' (safefilter' (>x) xs (f n)
        f = ???
```

04_Appendice/01_More_on_infinite_trees/**11_Infinite_Trees.lhs**

Thanks

Thanks to /r/haskell and /r/programming. Your comment were most than welcome.

Particularly, I want to thank Emm a thousand times for the time he spent on correcting my English. Thank you man.