Infinite Structures

It's 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 **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: **Haskell** allow it this code:

```
--29listaInfinita.hs
-- functions--------------------------------
--crea una lista de números = [1,2,..]
numeros :: [Integer]
numeros = 0:map (1+) numeros
--toma una cantidad de elementos de la lista
take' :: (Num a, Eq a) => a -> [b] -> [b]
take' n []      = []
take' 0 _       = []
take' n (x:xs) = x:take'(n - 1) xs
-- main program------------------------------
main = print . take' 10 $ numeros
```

```
Prelude> :l 29listaInfinita                                       enter
[1 of 1] Compiling Main  ( 29listaInfinita.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                       enter
[0,1,2,3,4,5,6,7,8,9]
```

How it stop?

Instead of trying to **evaluate the whole numeros function**, **only evaluates the elements when they are needed**.

Also, note that **Haskell** has **infinite lists notation**

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

Most of the **functions** can work with them. Also, there is a **take built-in function** that is equivalent to our own **take' function**.

Suppose we don't mind to have 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 you can control this **object** writing the following **function**:

```
-- take all element of a BinTree
-- up to some depth
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
```

This is the program:

```
--30arbolInfinito.hs
-- datatype------------------------------------------------
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
               deriving (Show, Eq, Ord)
-- functions-----------------------------------------------
--crea un árbol infinito
nullTree :: BinTree Integer
nullTree = Node 0 nullTree nullTree

--toma datos del árbol infinito hasta
--donde indica el valor de <n>
treeTakeDepth :: (Num a1, Eq a1) => a1 -> BinTree a -> BinTree a
treeTakeDepth _ Empty = Empty
treeTakeDepth 0 _     = Empty
treeTakeDepth n (Node x left right) =
```

```
  let
    nleft  = treeTakeDepth (n - 1) left
    nright = treeTakeDepth (n - 1) right
  in
    Node x nleft nright
-- main program-------------------------------------------
main = print . treeTakeDepth 4 $ nullTree
```

```
Prelude> :l 30arbolInfinito                                   enter
[1 of 1] Compiling Main ( 30arbolInfinito.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                   enter
Node 0 (Node 0 (Node 0 (Node 0 Empty Empty) (Node 0 Empty
Empty)) (Node 0 (Node 0 Empty Empty) (Node 0 Empty Empty)))
(Node 0 (Node 0 (Node 0 Empty Empty) (Node 0 Empty Empty))
(Node 0 (Node 0 Empty Empty) (Node 0 Empty Empty)))
```

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

```
iTree = Node 0 (dec iTree) (inc iTree)
  Where
    dec (Node x le ri) = Node (x-1) (dec le) (dec ri)
    inc (Node x le ri) = Node (x+1) (inc le) (inc ri)
```

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 the **function**:

```
-- apply a function to each node of Tree
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:

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

```haskell
--31arbolInfinito0.hs

-- datatype-----------------------------------------------
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
               deriving (Show, 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-----------------------------------------------
--crea un árbol infinito
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)
--toma datos del árbol infinito hasta
--donde indica el valor de <n>
treeTakeDepth :: (Num a1, Eq a1) => a1 -> BinTree a -> BinTree a
treeTakeDepth _ Empty = Empty
treeTakeDepth 0 _     = Empty
treeTakeDepth n (Node x left right) =
```

```
  let
    nleft  = treeTakeDepth (n - 1) left
    nright = treeTakeDepth (n - 1) right
  in
    Node x nleft nright
-- main program-------------------------------------------
main = print . treeTakeDepth 4 $ infTreeTwo
```

```
Prelude> :l arbolInfinito0                                        enter
[1 of 1] Compiling Main  ( arbolInfinito0.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                       enter
< 0
: |##-1
:     |##-2
:         |##-3
:         *##-1
:     *##0
:         |##-1
:         *##1
: *##1
:     |##0
:         |##-1
:         *##1
:     *##2
:         |##1
:         *##3
```

Hell Difficulty Part

Congratulations for getting so far! Now, some 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 there is a strange imperative-like notation for dealing with IO?***

Answers might be complex. But they all are very rewarding

Deal With IO

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
```

- **Each line type 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. You can't use **pure functions** here. To use **pure functions** you could do **action2 (pure function x)**.

In this section, I will explain **how to use IO, not how it works. You'll see how Haskell can separate pure functions from impure functions in two parts of the program**.

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

What to achieve?

Enter a number list, and print the sum of the numbers

```
--32io.hs
-- function
toList :: String -> [Integer]
toList input = read ("[" ++ input ++ "]")
--main program
main = do
  putStrLn "Entre numeros separados por coma:"
  input <- getLine
  print . sum . toList $ input
```

```
Prelude> :l 32io                                              enter
[1 of 1] Compiling Main            ( 32io.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                   enter
Entre números separados por coma:
1,3,5,7                                                       enter
16
```

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

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

We can note **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 about **(<-) symbol effect**.

```
do
 x <- something
 if something :: IO a then x :: a
```

Another important note about using **IO**: all lines in the **do block** must be one of the two forms:

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

```
or
```

```
value <- action2     -- where
                     -- action2 :: IO b
                     -- value   :: b
```

These two kinds of line will correspond to two different ways of **sequencing actions**. This sentence's meaning should be clearer later (next section).

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

```
*Main> main                                              enter
Entra una lista de números, separados por coma:
foo                                                      enter
*** Exception: Prelude.read: no parse
```

Argh! An evil **error message crash** the **program**. Our first improvement will simply be answering with a more friendly message.

In order to do this, we must detect that something was wrong. Here is a simple way to do this: **Maybe type**. This is a very **useful type** in **Haskell**.

What is this thing? Maybe is a **type constructor** that takes **one parameter**. Its definition is:

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

Maybe datatype is a nice way to show **there was an error** while trying to **create/compute** a **value**. A great example is **maybeRead function**. This **function** is similar to **read function**, but if something **is wrong** it will return a **Nothing value**. If the value is right, it returns **Just <value>**. Don't try to understand so much about this **function**. I use **reads function (lower level function)** instead **read function**.

```
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 **need to test** our **main function value**.

```
--33error.hs
--functions
maybeRead :: Read a => String -> Maybe a
maybeRead txt = case reads txt of
                  [(x,"")]    -> Just x
                  _           -> Nothing

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

--main program
main :: IO ()
main = do
  putStrLn "Entre números separados por coma:"
  input <- getLine
  let maybeList = getListFromString input in
    case maybeList of
      Just lista  -> print . sum $ lista
      Nothing -> error "formato malo. Adios."
```

Prelude> :l 33error	enter
[1 of 1] Compiling Main (33error.hs, interpreted)	
Ok, modules loaded: Main.	
*Main> main	enter

```
Entre números separado por coma:
5,3,6,2,8                                                   enter
24
*Main> main                                                 enter
Entre números separado por coma:
foo                                                         enter
*** Exception: formato malo. Adios.
CallStack (from HasCallStack):
  error, called at error.hs:20:26 in main:Main
```

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

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

A very important thing to note is the **type** of all the **functions** defined so far. There is only a **function** that contains **IO in its type**: **main**. ***It means that main function is impure***. But **main function** uses a **pure function: getListFromString**. So it's clear, just looking out **types signatures** what **functions are pure** and what **functions are impure.**

Why does purity matter? Among 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 often put as **most code** as possible inside **pure functions**

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

We keep the first part:

```
--34error.hs
--functions
maybeRead :: Read a => String -> Maybe a
maybeRead txt = case reads txt of
```

```
                  [(x,"")] -> Just x
                  _        -> Nothing

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

askUser :: IO [Integer]
askUser = do
  putStrLn "Entre números separados por coma:"
  input <- getLine
  let maybeList = getListFromString input in
    case maybeList of
      Just lista  -> return lista
      Nothing     -> askUser
--main program
main :: IO ()
main = do
  list <- askUser
  print . sum $ list
```

```
Prelude> :l 34error                                          enter
[1 of 1] Compiling Main          ( 34error.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                  enter
Entre números separados por coma:
foo                                                          enter
Entre números separados por coma:
1,hola,4                                                     enter
Entre números separados por coma:
1,2,3,4,5                                                    enter
[1,2,3,4,5]
```

Now we create a **function askuser** what will ask you for a list of integers, and repeat it until the **input is right**.

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** what **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**.

IO trick explained

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 (>>=)**.

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 "Entre números separados por comas:"
  input <- getLine
  let maybeList = getListFromString input in
    case maybeList of
      Just lista  -> return lista
      Nothing     -> askUser

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

First remark: this looks **imperative**. **Haskell is powerful enough to make impure code looks 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's **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 "Entre números separados por comas:"
  input <- getLine
  let maybeList = getListFromString input in
    case maybeList of
      Just lista  -> return lista
      Nothing     -> askUser
```

After:

```
askUser w0 =
    let (_,w1)     = putStrLn "Entre números separados por comas:" in
    let (input,w2) = getLine w1 in
```

```
    let (lis,w3)    = case getListFromString input of
                        Just lis -> (lis,w2)
                        Nothing  -> askUser w2
    in
        (lis,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
```

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 **blindBind function**:

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

I didn't simplify this definition for clarity purposes. 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.

```
--35sumaLista.hs

--pure functions
```

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

maybeRead :: Read a => String -> Maybe a
maybeRead texto = case reads texto of
                       [(x,"")] -> Just x
                       _        -> Nothing
--impure functions
askUser :: IO [Integer]
askUser = do
  putStrLn "Entra números separados por coma:"
  input <- getLine
  let maybeList = getListFromString input in
      case maybeList of
          Just lista -> return lista
          Nothing    -> askUser

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

```
Prelude> :l 35sumaLista                                       enter
[1 of 1] Compiling Main    ( 35sumaLista.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                   enter
Entra una serie de números separados por coma:
1,r,4,t                                                       enter
Entra una serie de números separados por coma:
1,2,3,4                                                       enter
10
```

Utilizando sintaxis do

```
--36sumaLista.hs
import Data.Maybe
--pure functions
```

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

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

--impure functions
askUser :: IO [Integer]
askUser =
  putStrLn "Entre números separados por coma:" >>
  getLine >>= \input ->
    let maybeList = getListFromString input in
      case maybeList of
        Just lista -> return lista
        Nothing    -> askUser

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

```
Prelude> :l 36sumaLista                                        enter
[1 of 1] Compiling Main    ( 36sumaLista.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                    enter
Entre números separados por coma:
1,2,3,r                                                        enter
Entre números separados por coma:
1,2,3,4                                                        enter
10
```

Utiliza: bind (>>=), then (>>) and return

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
```

You should **generally safely ignore** this **function (fail)**, which I believe exists for historical reasons

Remarks:

- *the keyword **class** is not your friend. A **Haskell** **class** is* not *a **class like 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**. If one **type** belongs to a **typeclass**, this **type** must meet with all **type class's functions**.*
- *In this particular **typeclass** example, 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

Monad's type class information provided on **ghci**

```
Prelude>:i Monad
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

enter

```
  fail :: String -> m a
  {-# MINIMAL (>>=) #-}
     -- Defined in ‘GHC.Base’
instance Monad (Either e) -- Defined in ‘Data.Either’
instance Monad [] -- Defined in ‘GHC.Base’
instance Monad Maybe -- Defined in ‘GHC.Base’
instance Monad IO -- Defined in ‘GHC.Base’
instance Monad ((->) r) -- Defined in ‘GHC.Base’
instance Monoid a => Monad ((,) a) -- Defined in ‘GHC.Base’
Prelude>
```

Maybe is a monad

There are a lot of different **types** that are **instances of Monad type class**. 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 **700€** only if you can afford the following operations **list** without your balance dipping below **zero**.

```
--37banco.hs
--pure functions
deposit, withdraw :: Num a => a -> a -> a
deposit value7 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
--impure functions
main = do
  print . eligible $ 300
  print . eligible $ 299
```

```
Prelude> :l 37banco                                                    enter
[1 of 1] Compiling Main       ( 37banco.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                            enter
True
False
```

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

```
--38banco.hs

--pure functions

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
--impure functions
main = do
  print . eligible $ 300
  print . eligible $ 299
```

```
*Main> :l 38banco                                                    enter
[1 of 1] Compiling Main           ( 38banco.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                          enter
Just True
Nothing
```

Not bad, but we can make it better

```
--39banco.hs
--pure functions
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
--impure functions
main = (print . eligible $ 300) >>
  (print . eligible $ 299)
```

```
Prelude> :l 39banco                                          enter
[1 of 1] Compiling Main          ( 39banco.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                  enter
Just True
Nothing
```

We have proven that **Monads** are a **good way to make our code more elegant**. Note this code organization's idea, 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 is a very simple example. We saw the utility of the **IO monad**. But now for a cooler example, lists.

The list monad

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

```
--40lista.hs

import Control.Monad (guard)
--value
allCases = [1..10]
--function
resolve :: [(Int,Int,Int)]
resolve = do
  x <- allCases
```

```
  y <- allCases
  z <- allCases
  guard $ 4 * x + 2 * y < z
  return (x,y,z)
--main program
main = do
  print resolve
```

```
Prelude> :l 40lista                                                     enter
[1 of 1] Compiling Main       ( 40lista.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                             enter
[(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**:

```
Prelude> allCases = [1..10]
Prelude> print $ [(x,y,z) | x <- allCases, y <- allCases, z <- allCases, 4*x + 2*y < z]
[(1,1,7),(1,1,8),(1,1,9),(1,1,10),(1,2,9),(1,2,10)]
```

I won't **list** all the **monads**, but there are many of them. We can simplifies several notions in **functional programming** using **monads**. Particularly, **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!

Appendix

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

More on Infinite Tree

In the section **Infinite Structures** we have been seeing some simple constructions. Unfortunately we have not considered two properties from our tree:

1. **no duplicate node value**
2. **an 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**:

```
--41arbolInfinito.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)
```

```haskell
      replace c new string =
        concatMap (change c new) string
        where
          change c new x
              | x == c = new
              | otherwise = x:[]

-- pure functions-----------------------------------------------------
shuffle :: Integral b => [b]
shuffle = map (\x -> (x * 3123) `mod` 4331) [1..]

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

treeTakeDepth :: (Num a1, Eq 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

-- impure function----------------------------------------------------
main = do
  putStrLn "lista de 10 números pseudo aleatorio:"
  print $ take 10 shuffle
  putStrLn "\ntreeTakeDepth 4 (treeFromList shuffle):"
  print . treeTakeDepth 4 $ treeFromList shuffle
```

```
Prelude> :l 41arbolInfinito                                          enter
[1 of 1] Compiling Main  ( 41arbolInfinito.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                          enter
lista de 10 números pseudo aleatorio:
```

```
[3123,1915,707,3830,2622,1414,206,3329,2121,913]

treeTakeDepth 4 (treeFromList shuffle):
< 3123
: |>>1915
:    |>>707
:       |>>206
:       *>>1414
:    *>>2622
:       |>>2121
:       *>>2828
: *>>3830
:    |>>3329
:       |>>3240
:       *>>3535
:    *>>4036
:       |>>3947
:       *>>4242
```

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

For example:

```
*Main> treeTakeDepth 4 (treeFromList [1..])
< ^CInterrupted.
*Main>
```

enter control <c>

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

Nonetheless, it's still a very cool example of what **non strict programs** have to offer

A good exercise to do:

- Prove the existence of a **number n** so that **treeTakeDepth n (treeFromList shuffle)** will enter an **infinite loop**. **Answer = 8**
- Find an **upper bound** for **n**. **Answer = 4331**

- Prove if there is not a **shuffle list**, for any **depth**, the program ends.

In order to solve these problem we will slightly modify our **treeFromList** and **shuffle function**.

A **first problem**, is the **lack of infinite different number** in the **shuffle function** implementation. We generated only **4331 different numbers**. To solve 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 following property: have not an **upper** or **lower bound**. But having a better **shuffle list** isn't enough to enter in an **infinite loop**.

Often, we cannot decide whether **filter (<x) xs** is **empty**. Then to solve 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 branch must all be strictly inferior to the label of the root***
- ***any element of the right branch must all be strictly 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 function**. We simply have replaced **filter** by **safefilter function**.

```
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 safefilte**r is almost equivalent to **filter** but **don't enter infinite loop** if the result is a **finite list**. If it can't 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 lis = safefilter' f lis 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**.

```
--42arbolInfinito2.hs

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

-- typeclass instance------------------------------------------
{-
instancia de BintTree en la clase de tipo Show
```

```
permite mostrar un árbol en forma más gráfica
-}
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:[]

-- pure functions---------------------------------------------------
{-
pseudo random number list
-}
shuffle :: [Integer]
```

```haskell
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

--crea una árbol binario a partir de una lista
treeFromList :: (Ord a) => [a] -> BinTree a
treeFromList []     = Empty
treeFromList (x:xs) = Node x (treeFromList (safefilter (<x) xs))
                             (treeFromList (safefilter (>x) xs))
{-
safefilter reemplaza a la función filter
lo cual permite manejar estructures de
listas infinitas hasta 1000 pasos
-}
safefilter :: (a -> Bool) -> [a] -> [a]
safefilter f lis = safefilter' f lis nbTry
  where
    nbTry = 1000
    safefilter' _ _ 0      = []
    safefilter' _ [] _     = []
    safefilter' f (x:xs) n =
      if f x
        then x:safefilter' f xs nbTry
        else safefilter' f xs (n - 1)

{-
crea un árbol binario con una
profundidad cuyo valor expresa <n>
-}
treeTakeDepth :: (Num a1, Eq a1) => a1 -> BinTree a -> BinTree a
treeTakeDepth _ Empty = Empty
treeTakeDepth 0 _     = Empty
treeTakeDepth n (Node x left right) =
```

```haskell
  let
    nl = treeTakeDepth (n - 1) left
    nr = treeTakeDepth (n - 1) right
  in
    Node x nl nr

-- impure function------------------------------------------------------
main = do
  putStrLn "lista de 10 números pseudo aleatorios:"
  print $ take 10 shuffle
  putStrLn "\ntreeTakeDepth 2 (treeFromList shuffle):"
  print . treeTakeDepth 2 $ treeFromList shuffle
```

```
Prelude> :l 42arbolInfinito                                          enter
[1 of 1] Compiling Main  ( 42arbolInfinito.hs, interpreted )
Ok, modules loaded: Main.
*Main> main                                                          enter
Lista de 10 números pseudo aleatorios:
[272,-248,501,27,-448,306,-213,469,-154,396]

treeTakeDepth 3 (treeFromList shuffle):
< 272
: |>>-248
:     |>>-448
:     *>>27
: *>>501
:     |>>306
:     *>>527
Main*>
```

It's an exercise to do by 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:

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

- 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'** shuffleis infinite. And prove it. Disclaimer, this is only a conjecture.

```haskell
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 = undefined
```

The END