Functional Programming – The complete works

Josh Felmeden

2018
December

Contents

1 Introduction to Lists

Lists are a really useful part of Haskell, because they are a data structure that only take one type of data. This means that we have a data structure where we can process each entity in the same way. The word for this is **homogenous**. There are a number of operations that we can perform on a list. They are (but not limited to):

```
-- Appending
ghci> [1,2,3,4] ++ [5,6,7,8]
[1,2,3,4,5,6,7,8]

-- Cons
ghci> 1:[2,3,4]
[1,2,3,4]

-- BangBang
ghci> [1,2,3,4,5,6] \!! 2
3

--Removing elements
ghci> head [1,2,3,4,5]
1

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

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

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

There is, of course, the length function that returns the length of the list.

This is a pretty basic introduction to lists, but you pretty much know how they work anyway, so I'll leave it there.

2 Folds

Oh god. *Folds*. Recursion is one of the most dangerous (and useful) constructs in the programming world. It's really easy to create an accidental infinite loop and therefore black hole with them because of the terminating factor can not be reached. Folding however, is quite predictable. We perform a fold onto a list in the following way:

```
ghci> foldr (+) 1 [3,5,7,9]
25
```

Because, we can write a list like so: 3:5:7:9:[] , and so if we replace the cons with the monoid that we fed into the fold, we get $3 + 5 + 7 + 9$ and then because it's fold**r**, we put the argument on the right, leaving us finally with: $3 + 5 + 7 + 9 + 1 = 25$

Now, we know (hopefully) that monoids have a type of a $->$ b $->$ b, which means that our foldr function has type:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

All that's left to do is to show how it works which is as follows:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
```

And that pretty much sums up foldr. The important things to remember are:

- The type is (a -> b -> b)-> b -> [a] -> b
- It's really important to remember that the argument ends on the *right* for foldr, otherwise things like subtraction get really messy. It's better to draw out the brackets and not be lazy and skip ahead.

Just as an example, let's evaluate sum in terms of foldr.

```
ghci> sum [3,1,4]
--={sum}
foldr (+) 0 (3:1:4:[])
--={foldr}
(+) 3 (foldr (+) 0 (1:4:[]))
--={foldr}
(+) 3 ((+) 1 (foldr (+) 0 (1:4:[])))
--={foldr}
(+) 3 ((+) 1 ((+) (foldr (+) 0 (4:[]))))
--={foldr}
(+) 3 ((+) 1 ((+) 4 ((+) (foldr (+) 0 ([])))))
--={foldr}
(+) 3 ((+) 1 ((+) 4 + 0))
={reverse Polish notation}
3 + (1 + (4 + 0))
={+}
8
```

3 Newtypes, types, and data

There are multiple ways of introducing new data types in Haskell.

3.1 Data

Here, we'd write something along the lines of:

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

This is a shorthand for the following:

```
data Maybe a where
    Nothing :: Maybe a
    Just :: a -> Maybe a
```

Which introduces the type Maybe a for any a, along with the constructors 'Nothing' and 'Just'. Remember, all constructors start with a capital letter, and they result in the type being defined, and a pattern being matched on.

Another example that we can see is:

```
data Either a b where
    Left :: a -> Either a b
    Right :: b -> Either a b
```

The two keywords Left and Right have capital letters because they are constructors and not functions so no function body is required.

3.2 Type Synonyms

A type synonym introduces a name for an existing type. Instead of writing [Maybe a] we might want to write Maybes a instead. To achieve this, we just write:

```
type Maybes a = [Maybe a]
```

3.3 Newtypes

Units of measure sometimes store a data representation, but should actually be kept distinct. What I mean by this is that the number '5' can have a number of meanings depending on the context. It could mean 5 centimeters, or meters, or even monkeys. To differentiate between the things, we can use something called a **newtype** in Haskell. To use this, all you do is:

```
--This is not valid Haskell
newtype Metre where
    Metre' :: Int -> Metre
```

This introduces a new constructor called Metre' which takes in a Int and makes a value of type Metre.

3.4 Type classes

A type class is a way to give functions multiple meanings depending on the type of the function. For example, the function print prints values to the screen, and can have multiple definitions depending on the type that you feed it, because it can do all of the following:

```
show True      = "True"
show 5         = "5"
show (Just 3)  = "Just 3"
show "hello"   = "hello"
```

It would be understandable to think that the function 'show' has the type:

```
show :: a -> String
```

You would of course be completely wrong, but it is understandable why you thought that. Instead, we introduce a family of show functions by creating a new type class:

```
class Show a where
    show :: a -> String
```

This means that the functions show now exists, with type:

```
show :: Show a => a -> String
```

Oh, by the way Show a is not a parameter, and => is not a function. All it means is 'if a is a member of the Show family, then show has type a−>string'.

The class definition gives the types of its functions, but doesn't give their implementation. We need to use an instance for each member of the family. For example, if we want to be able to print Bool values, then we write:

```
Instance Show Bool where
    show :: Bool -> String
    show True = "True"
    show Fals = "False"
```

This introduces the show that works for Bool values. Haskell actually has Show built in, and you can automatically derive functions for it. All you do is:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving Show
```

The deriving bit tells Haskell to make the appropriate instance of show for us. You don't get that with C, now do you?

3.5 Equality

In the same way that you need to tell Haskell to auto derive the show function, you need to do the same for equality. Take this as an example:

```
ghci> 5 == 3
False

ghci> 3 == 3
True
```

You couldn't compare two different types (like 5 == True), because they wouldn't make sense. To introduce this to our types, we do:

```
class Eq a where
    (==) :: a -> a -> Bool
```

And just like with show, we now have a function:

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

Neat! And that's pretty much it for the different data types.

4 Monoids

Hey, remember earlier on when I was talking about monoids and that you should know what they mean? Yeah, I was getting ahead of myself. We're now going to delve into the sweet depths of the monoid.

A monoid is an operation $\oplus$ together with a neutral element ϵ such that the following laws hold true:

$$\begin{aligned} \text{associativity} &: (x \oplus y) \oplus z = x \oplus (y \oplus z) \\ \text{left unit} &: \epsilon \oplus y = y \\ \text{right unit} &: x \oplus \epsilon = x \end{aligned}$$

Formally, a monoid is described as a triple $\langle X, \oplus, \epsilon \rangle$, and can be seen in the example:

$$\begin{aligned} \langle \mathbb{N}, +, 0 \rangle &\text{ addition} \\ \langle \mathbb{N}, \times, 1 \rangle &\text{ multiplication} \end{aligned}$$

Each monoid consists of three things. The *set*, the *function*, and the neutral element. There are, of course, other monoids, but I can't be bothered to type them out, so I'll leave that one up to you.

We can actually capture the general pattern of a monoid in the following type class:

```
class Monoid m where
    mempty :: m --This is the equivalent of epsilon
    mappend :: m -> m -> m --This is the equivalent of the function
```

We consider an instance of this class to be valid when the three monoid laws hold true:

1. mappend x(mappend y z) = mappend(mappend x y) z
2. mappend mempty $y = y$
3. mappend x mempty $= x$

For example, we can write an instance where m = Int for addition:

```
instance Monoid Int where
    --emempty :: Int
    mempty = 0

    --mappend :: Int -> Int -> Int
    mappend = (+)
```

The problem with this is that now we have trapped ourselves into making (+) the monoid. What if we wanted *? Well, we can resolve this by using a newtype for each conflicting monoid.

To do sum properly,

```
newtype Sum = Sum Int

instance Monoid Sum where
    --mempty :: Sum where
```

```
    mempty = Sum 0
    --mappend :: Sum -> Sum -> Sum
    mappend (Sum x)(Sum y) = Sum (x+y)
```

Suppose we have a list of monoidal values and we want to collapse it (so $[x, y, z]$ into $x \oplus y \oplus z$), we can achieve this using a foldr.

```
fold :: Monoid m => [m] -> m
fold = foldr mappend mempty
```

5 Data Structures

One of the absolute **best** structures is the Maybe data type. It was defined as follows:

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

This introduces two different functions to construct:

```
Nothing :: Maybe a
Just :: a -> Maybe a
```

We can think of these constructors as boxes, where we have a box with nothing, and a box with Just, that contains the set a. One thing that we can do is to transform the contents of a Maybe a so that the values of type 'a' become types of 'b' instead. This would make a value of type 'Maybe b'.

```
mapMaybe :: (a->b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

5.1 Trees

Ok, now we really are in business here. Trees are kinda more complicated, because it's made up of node and leaf values. It kinda looks like this:

To define it properly in Haskell, we do this:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

Which creates two constructors that we can pattern match on in the normal way:

```
Leaf :: a -> Tree
Fork :: Tree a -> Tree a -> Tree a
```

Here are some values that construct trees:

```
Leaf True :: Tree Bool

Leaf 5 :: Tree Int

Fork (Leaf True) (Leaf False) :: Tree Bool

Fork (Leaf 3) (Leaf 5) :: Tree Int

Fork (Fork (Leaf 7) (Leaf 8)) (Leaf 9) :: Tree Int
```

However, you can't mix and match data types. This isn't a pick and mix, you know.

There are some more interesting types to consider, such as Tree (Maybe Int). If we had a tree like that, we can now have:

```
Leaf Nothing :: Tree (Maybe a)
Leaf (Just 4) :: Tree Maybe (Int)
Fork (Leaf Nothing) (Leaf (Just 7)) :: Tree (Maybe Int)
```

We can convert between trees in just the same way as we do with lists or maybes:

```
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Fork l r) = Fork (mapTree f l) (mapTree f r)
```

If you notice, Haskell's trees aren't the same trees as other languages, because it only has data in the leaves, rather than at every location. We're going to look at some variations of the tree now that will show us a more 'classic' tree, if you will.

5.1.1 Other types of tree

A similar structure is a Bush. The idea of this is that the values are in the nodes, and leaves don't exist. The code for this looks something like this:

```
data Bush a = Tip a
            | Node (Bush a) (Bush a)
```

If I were to draw it using a visual, it'd look like this:

```
Tip                      Node
                        /  |  \
                     Tip   a   Tip
```

Here are some values that use this structure:

```
Tip :: Bush a
Node Tip 8 Tip :: Bush Int
Node Tip False Tip :: Bush Bool
Node (Node Tip 5 Tip) 8 (Node Tip 3 Tip)
```

We have created the following constructors:

```
Tip :: Bush a
Node :: Bush a -> a -> Bush a -> Bush a
```

Just like before, you can map over the contents of a Bush.

```
mapBush :: (a -> b) -> [a] -> [b]
mapBush :: f Tip = Tip
mapBush f (Node l x r) = Node (mapBush f l) (f x) (mapBush f r)
```

That's all from tree friends. (I'm so sorry).

6 Functors

What the hell is a functor? Well, if we look at all the different maps we've looked at so far in this essay (?), they all look kinda samey:

```
map ::      (a -> b) -> [a] -> [b]
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapTree ::  (a -> b) -> Tree a -> Tree b
mapBush ::  (a -> b) -> Bush a -> Bush b
```

The generalisation of data structures like [], Maybe, Tree, Bush is called a **functor**. We can use a type class to generalise this:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Oh, f is **NOT** a function here. It's actually a type. f could be []. Maybe, Tree or even Bush. A valid instance of the functor class has to obey these laws:

1. fmap id = id (This is the identity law)
2. (fmap g).(fmap f) = fmap(g.f) (This is the composition law)

These laws make use of the function fmap that we are defining and the functions id and (.).

They are actually already defined as:

```
id :: a -> a
id x = x

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g.f) x = g (f x)
```

7 Structural Induction

We can prove statements by using a thing called induction. You know, the one from maths? Yeah we can transfer that one over to computer programming. Talk about multi-tasking revision!

In terms of the computing version of induction, we have to prove the statement for the empty case. This is the equivalent of the *base case* in maths. After this, we have to prove it for the recursive case (assuming the truth for the smaller cases). For lists, we have:

```
instance Functor [] where
    --fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```

8 Indexed trees

It is possible to extract indexed elements from a list. We've actually already discussed this, it's above entitled 'bang bang'. To do this with trees, we need to redefine the data type tree, where it contains information about how many elements it has at the nodes. We shall name it ITree.

```
        4
       / \
      2   2
     / \ / \
    a  b c  d
```

To define it, it's going to look like this:

```
data ITree a
    = ILeaf a
    | Inode Int (ITree a) (ITree a)
```

To get a feel for how this corresponds to lists, we need to define a function that flattens a tree. That means that it converts the tree into a flat list, so in the example above, it would turn into [a, b, c, d].

At the moment, we don't have a way of creating an ITree that contains no values of type 'a'. In other words, we can't flatten to an empty list. We can easily add a constructor to do that for us.

To make an 'ITree a' we need to make a 'mkITree'.

```
mkITree :: [a] -> ITree a
mkITree [] = error "Please don't do that again"
mkITree [x] = ILeaf x
mkITree xs = INode ix l r
    where ix = length xs
    l = mkITree (take (ix `div` 2) xs)
    r = mkITree (drop (ix `div 2) xs)
```

We can make this better so that we don't have to use take and drop, and instead use the 'splitAt' function defined in the prelude.

```
mkITree xs = INode ix l r
    where ix = length xs
        (ys, zs) = splitAt (ix `div` 2) xs
        l = mkITree ys
        r = mkITree zs
```

I'm not going to write out the splitAt function because I'm lazy and it's not that interesting.

Now, we're all ready to retrieve elements from the tree that we have made. First of all, we need to start with a spec:

```
retrieve :: ITree a -> Int -> a
retrieve t n = flatten t !! n
```

This is a good specification because we are defining behaviour in terms of known functions and another data type that we know behaves nicely.

That's really lazy though, and we're not about that life, so here's a better implementation with a pattern match:

```
retrieve :: ITree a -> Int -> a
retrieve (ILeaf x) 0 = x
retrieve (Inode ix l r)
    = case l of
        ILeaf x -> x
        INode ixl ll lr -> retrieve ll 0
retrieve (Inode ix l r) n
    = case l of
        ILeaf x -> retrieve r (n-1)
        INode ixl lr ->
            if n < ixl then retrieve l n
                       else retrieve r (n - ixl)
```

What a monster of an equation. In the above example, if we wanted to find the second element, then that would be 'd'. When we look at the tree, we know that the left side has 2, and the right side has 2. Since $4 > 2$, we need to move to the right hand side of the tree. We're left with $4 - 2 = 2$, so we take the second element to give us the right answer of 'd'.

The case statements in the code are there because the information that we needed is not actually available in the node. It would be better to store the information about the length of the tree on the left, so we know what we're dealing with, but we have to wait for the result of the next tree length, so that is why we use if.

It'd be nice if you could come up with a better structure that stores the index information that we need in the previous tree so we don't need to use the case value.

9 Folding data structures

The foldr function was useful to allow us to define many functions on lists. It is also possible to define a fold for any data structure that uses algebra (like sum/products etc.). To do this, we have to actually understand how we might have got this function in the first place.

We all know the foldr function definition: (a −> b −> b)−> b −> [a] −> b, and from this, we can go onto the type for Tree a. Recall that there are two things that we have in a tree, the Fork and the Leaf:

```
Leaf :: a -> Tree a
leaf :: a -> b

Fork :: Tree a -> Tree a -> Tree a
fork :: b -> b -> b
```

I actually decided to slip two MORE constructors in: leaf and fork. It's just by adapting 'Leaf' and 'Fork'. These will be the parameters for the fold:

```
foldTree :: (a -> b) -> (b -> b -> b) -> Tree a -> b
foldTree leaf fork (Leaf x) = leaf x
foldTree leaf fork (Fold l r)
    = fork (foldTree leaf fork l) (foldTree leaf fork r)
```

To use this fold tree, we do a similar thing as foldr. For example, suppose we have a tree that is full of numbers and we want to sum them up:

```
      /\
     /\ 3
    1 2
```

This tree is represented by the following values:

```
Fork (Fork (Leaf 1) (Leaf 2)) (Leaf 3)
```

To sum the values in this tree, we just gotta use the sumTree function.

```
sumTree :: Tree Int -> Int
sumTree = foldTree leaf fork where
    leaf :: Int -> Int
    leaf x = x

    fork :: Int -> Int -> Int
    fork :: x y = x + y
```

How does this work? Well, we can show this by using a simple example.

```
sumTree (Fork (Fork (Leaf 1) (Leaf 2)) (Leaf 3))
-- ={sumTree}
foldTree leaf fork (Fork (Fork (Leaf 1) (Leaf 2)) (Leaf 3))
-- ={foldTree}
fork (foldTree leaf fork (Leaf 1) (Leaf 2)) (foldTree leaf fork (Leaf 3))
-- ={fork}
(foldTree leaf fork (Leaf 1) (Leaf 2)) + (foldTree leaf fork (Leaf 3))
-- ={foldTree}
(fork (foldTree leaf fork (Leaf 1) (foldTree leaf fork (Leaf 2))) +
(foldTree leaf fork (Leaf 3))
-- ={fork}
foldTree leaf fork (Leaf 1) + foldTree leaf fork (Leaf 2) + foldTree leaf
fork (Leaf 3)
-- ={foldTree xs}
Leaf 1 + Leaf 2 + Leaf 3
-- ={Leaf}
1 + 2 + 3
-- ={math}
6
```

We can demonstrate 'foldTree' by taking the size of a tree, which is the number of values that it contains:

```
sizeTree :: Tree a -> Int
sizeTree = foldTree leaf fork where
    leaf :: a -> Int
    leaf x = 1

    fork :: Int -> Int -> Int
    fork x y = x + y
```

Finally, there's also a property of the tree is height.

```
heightTree :: Tree a -> Int
heightTree = foldTree leaf fork where
    leaf :: a -> Int
    leaf x = 1

    fork :: Int -> Int -> Int
    fork x y = 1 + x `max` y
```

The main point of this is that by using the foldTree function, we don't have to worry about the repetitive structure of the code that does the recursion.

10 Lookup Maps

A really useful data structure is Map k v, which is kind of like a dictionary or an address book that gives values of type 'v' when keys of type 'k' are supplied. Basically, we kind of have this:

$$\text{Map } k\ v \approx k \to v$$

This says that maps and functions are interchangeable. The constructors in a map remain abstract because they can't be accessed directly. Abstract data types like this are useful for a lot of reasons. For instance, keeping a type abstract allows programmers to offer different underlying implementations at different points in time without end-users even knowing. Those silly people.

To import this data type, it's easy to do:

```
import Data.Map
```

This imports all of the functionality of the maps from an external module. It's kinda like #include in C. One of the functions that this imports like lookup :: Map k v -> k -> Maybe v. The problem is that lookup is a different function to the one that is imported by default by Prelude (by default), which is this one: lookup :: [(a,b)] -> a -> Maybe b. We need a way to tell the difference between the two. To do this, we just import Map in a different way:

```
import qualified Data.Map as M
```

Here, the M is arbitrary, and we chose M because it makes sense. Qualified means that we want explicit names. Because we imported it in a qualified way, we can refer to the Data.Map.Lookup as M.lookup. The other technique that you can do is to import Prelude, but then hide lookup:

```
import Prelude hiding (lookup)
```

But then you lose the functionality of the other lookup function.

The most simple map in the whole world is given by:

```
empty :: Map k v
```

This is a map that does not contain any entries. We also have a way of inserting elements into a map:

```
insert :: Ord k => k -> v -> v -> Map k v -> Map k v
```

The result of 'insert k v' is to insert v at the key k in map m. For example:

```
insert "Joe" 8 empty :: Map String Int
```

This contains a key called 'Joe', and it has the value 8. If there is already a value of Joe, then it overwrites the previous value that Joe contained. The 'lookup' function fetches values from a map. The result of lookup k m is 'nothing' if 'k' is not in the map 'm'. If not, the result is 'Just v', where v is the value that is associated with the key.

11 Trie Maps

A trie is a kind of tree that is basically a tree x map. The keys are just a list of elements. For example, if we have Trie Char Int, the Char are the edges and the Int are the nodes. I can't really draw one of these out because they're kinda complicated to do, so if you need one, then you need to go and look at the lecture notes (I think it's week 7 part 1) (It's not it's part 2 (second lecture of that week)).

A value of 'Nothing' is used if there are no entries that correspond to the key and a value of 'Just 0' because it means that there is no value that exists. A Trie is implemented as follows:

```
data Trie k v = Trie (Maybe v) (Map k (Trie k v))
```

The 'Maybe v' corresponds to the value of each node, while the other map corresponds to the link from this node to the others. Let's look at an individual tile:

```
        5
       /|\
      a v q
     /  |  \
    t1  t2  t3
```

This corresponds to:

```
Trie (Just 5) m
```

where m is the map:

$$'a' \mapsto t_1$$
$$'v' \mapsto t_2$$
$$'q' \mapsto t_3$$

Once we get an empty Trie (see above), then we can add elements to it. This is the function that we use:

```
insert :: Ord k => [k] -> v -> Trie k v -> Trie k v
insert [] v (Trie mv tkv) = Trie (Just v) tkv
insert (k:ks) v (Trie mv tkv) = Trie mv tkv'
    where
```

```
        tkv' :: Map k (Trie k v)
        tkv' - case Map.lookup tkv of
    Nothing -> Map.insert k (insert ks v empty) tkv
    Just t -> Map.insert k (insert ks v t) tkv
```

Christ, that's complicated. In the case where the keys are empty, we decide to ignore the value of 'mv' and replace it with 'Just v'. This is a destructive update and we will fix it later with the adjust function.

Once we have our beautiful trie, we want to be able to lookup values:

```
lookup :: Ord k => [k] -> Trie k v -> Maybe v
lookup [] (Trie mv tkv) = mv
lookup (k:ks) (Trie mv tkv) =
    case Map.loookup tkv
        Nothing -> Nothing
        Just t -> lookup ks t
```

The adjust function can be implemented in a dumb way by combining insert and lookups:

```
adjust :: Ord k => [k] -> (Maybe v -> Maybe v) -> Trie k v -> Trie k v
adjust ks f t = case lookup ks t of
    Nothing -> insert ks (f Nothing) t
    Just v -> insert ks (f (Just v)) t
```

We could even simplify this:

```
adjust ks f t = insert ks (f(lookup ks t)) t
```

The problem of this is that insert requires a value of the type 'v', but we are supplying a type 'Maybe v'. The actual adjust function needs to be able to both insert and delete values depending on the function f. There is an implementation, but that's on the Trie.lhs file.

12 Inputs and outputs

A computer program usually interacts with the outside world. Programs are no exception to this. The type 'IO' represents the computations that perform input/output in some way and returns some value. One form of I/O is reading files. In Haskell, this is achieved like so:

```
readFile :: FilePath -> IO String
```

So, FilePath is the name of the file, and the IO bit is the fact that Haskell is interacting with the 'real world' so to speak. The String value is the pure value result. Another useful function is one that allows us to write values to a file:

```
writeFile :: FilePath -> String -> IO()
```

This function takes in a filePath and a String and outputs the string to the file. By the way, filePath is a synonym for String:

```
type FilePath = String
```

The functions become useful in the context of a given program. Even the actual program of Haskell is given by the function is IO:

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

This is the main entry point for a program.

Two of the most useful functions that perform I/O are:

```
putStr :: String -> IO ()
```

This function takes a String and prints it to the terminal. One variation of this is:

```
putStrLn :: String -> IO ()
```

These functions produce output and the opposite of this is:

```
getLine :: IO (String)
```

In order to do these things in sequence we need a way to write several statements.

12.1 Do notation

The do notation allows different IO actions to be put into sequence. If we wanted to write 'Hello world!' to the terminal, we do:

```
do putStr "Hello "
   putStrLn "world!"
```

This exactly the same as:

```
do {putStr "Hello "; putStrLn "world!"}
```

We also have notation that allows us to extract values from an IO computation:

```
do putStrLn "What is your name?"
   name <- getLine
   putStrLn("Yourname is " ++ name ++ "idiot")
```

This has introduced a variable called 'name' which is based to the value of 'getLine'.

13 Sequencing

Do notation is apparently 'syntactical sugar' which means that it is translated to some more fundamental operations. They are as follows:

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

The result of 'p ¿¿ q' is to execute 'p' and then result in the value of 'q'. For example.

```
putStr "Hello " >> putStrLn "world!"
```

This executes one computation follwed by another but there can be no data from the first computation that is used by the second. The value of type 'a' is discarded.

A more useful combinator is called bind. It is defined as follows:

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

Given some computation p :: IO a and a function f :: a -> IO b the result of p >> f is to first execute and then feed the result to the function f and perform the resulting computation. An example of this is:

```
getLine >>= printName
```

We're gonna define printName right now:

```
printName :: String -> IO ()
printName xs = putStrLn("you name is " ++ xs)
```

The result of the example is to read a line of input from the user and print it straight back out. The translation form do notation by using these two bad boys is:

```
do p
   q
```

is the same as

```
p >> q
```

And:

```
do x <- p
f x
```

is the same as

```
p >>= f
```

Wild eh? Suppose we want to use out 'printName' function. We got two options:

```
printName "foolish"
```

or

```
return "Foolish" >>= printName
```

Here, return :: a -> IO a will take a value of type 'a' and bundle it up in a nice little IO package.

14 IO in the world

One way to understand 'IO a' is that it is an abstraction that takes the state of the world and produces a new world along with a value of type a.

It kinda looks like this:

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

For instance we can think of putStr"Hello":

```
putStr "hello" :: IO ()
=
putStr "hello" :: World -> (a, World)
```

The first world is our world, and the second one is the new world, where 'hello' has been written to the screen.