Functional Languages Recap

Riccardo Montagnin

Academic Year 2017/18

Abstract

This document represents a small recap of all the things that have been taught during the Functional Languages course by Professor Filè.
All the written things may contain some errors, so please do not trust every single bit of information coming from them but feel free to do your own search if something may look fishy. Also, if you find any error please contact me or feel free to correct it inside this file too.
In order to write this recap, I looked first into the prof. Filè's lessons slides, and then added some missing information coming from the course book, *Programming in Haskell - Graham Hutton (2nd edition)*, and the "Learn You a Haskell for Great Good!" site.

Contents

Chapter 1

Introduction

1.1 Useful resources

During the *Functional Languages* course the main focus is the Haskell programming language. It is uses in order to explore the "joys" of the functional approach when coding a program, to learn about pure and lazy languages, to learn about type inference and how it can be useful when done automatically, and also to see how Haskell is compiled and learn something about run-time management.
The main reference of the entire course is the *Programming in Haskell* book, written by Graham Hutton and published by the Cambridge University Press. Also, another useful book is *Lean you a Haskell for great good* by Miran Lipovaca, which can be found at the "Learn You a Haskell for Great Good!" site.
Other useful documentation may be found at the Course CIS194 site by the University of Pennsylvania and inside the *Real World Haskell* book by Bryan O'Sullivan, Don Stewart and John Goerzen, which can be found at its official site.

1.2 Historical context

During the years 1930, the main problem of computer science was to define with precision what computable functions were. In order to do this, there were two main approaches.

- The **Turing** approach, which defined the model of what is now called a *computer* with a mutable memory
- The **Church and Curry** approach, which created the rules used to define and compose functions inside which all is immutable. This particular approach was called *lambda calculus*

Later on it was demonstrated that both approaches define the same set of functions that are therefore the so called **computable functions**.

From both approaches may programming languages were then created. Same examples may be *FORTRAN, Pascal, C, C++, Java* from the Turing approach, and *LISP, ISWIM, Miranda, Haskell* from the Church and Curry approach.
During recent years some programming languages were also created in order to use both languages paradigms. Some examples may be *Scala, Ruby* and *Occaml.*

1.3 A taste of Haskell

As seen before, Haskell is a functional language which strictly adheres to the Church and Curry definition. Inside it all variables are functions, and everything is immutable.
Let's get a taste of it by simply looking at some basic functions that can be defined.

```
--| Contents of sum.hs
--| Sums a list of numbers
sum[ ] = 0
sum(n:ns) = n + sum ns
```

If we then type the command

```
>> ghci sum.hs
Prelude> sum[1, 2, 3]
```

we get the correct result, 6.
We can also check the type of sum by using

```
>> ghci sum.hs
Prelude> :t sum
```

which should return `Num a => [a] -> a`.
This signature tells us that sum works only with types that are contained inside the type class Num (briefly, they are numbers), takes one input which is a list of Num (the [a] signature) and returns a single value of type Num (a).
The main point to observe here is that Haskell supports many types of numbers integers and floating point, and groups all of them inside the class Num which can then be used to define more generic functions.
It's also interesting to see that the type of the functions sum if inferred by the compiler just looking at the type of 0 and +.

Even with this small example we can see that functional languages are pure, and so don't have storable and/or mutable variables. They only have names that hold a constant value and instead of changing the values of variables new values are computed. Also, functions do not produce side effects whatsoever.
The main advantages of this approach is that we can get a higher level of abstraction and simplicity, we can be more correct and have easier parallel implementation of our programs. The disadvantages, however, are a difficult relation with the I/O actions and a difficult way to handle exceptions.

1.4 The Glasgow Haskell Compiler

Following the sire Haskell.org you can find the Glasgow Haskell Compiler (shortly called GHC) which contains all the stuff used in order to compile and/or interpret a program written using Haskell.
Once installed the command seen above, ghci, runs the interpreter and takes you inside Prelude, an environment where you can type expressions that are immediately executed.
It's helpful to know that Prelude is in reality a library and contains a lot of built-in functions that can be useful when coding really long programs. Some functions are

- head which takes the first element of a list
- tail which takes all the elements of a list starting from the 2nd to the last one
- take n which takes the first n elements of a list
- drop n which deletes the first n elements from a list
- length which returns the size of a list
- sum which sums all the elements inside a list

Chapter 2

Types and Type classes

2.1 Type checking principles

Inside Haskell a type of a collection of values. As an example, we can see the type `Bool` which is the collections that can be seen as

```
Bool -> {True, False}
```

Also, the notation `Bool -> Bool` represents the set of functions that go from a `Bool` value to another `Bool` value.

In Haskell type inference is done at compile time, thus, before evaluating. With this method, Haskell can ensure to be **type safe** in the strong sense that during evaluation **no type error can occur**.
In general, type checking can be done both statically or during evaluation. We can define the meaning of *type safeness* by saying that if a program is **type safe** than **all the type errors are exposed sooner or later**.
An important thing to remember is to not fall in the trap of believing that in a type safe language no error can occur. Type safeness assures that all the type errors are exposed, but type errors can still be done by programmers and other sort of errors can still occur.
An important drawback of Haskell is that the following expression

```
if True then 1 else False
```

even if during execution it always goes inside the `then` branch, is **not correct** as the `then` and `else` branches have different return types. So, if we could execute this code everything would run correctly, but we cannot do this in Haskell as the compilation fails before executing due to a type error.

2.2 Types

2.2.1 Basic types

Inside Haskell we have some basic types that represents the foundation of the languages. Those are `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` and `Double`.
We also have lists types, which represent sequences of values of the same type. For example we can see that

```
[False, True, False] :: [Bool]
["One", "Two", "Three"] :: [String]
```

An important thing to notice is that a list type does not convey the length of the list itself. We can then have `[[String]]`, `[[[String]]]` and so on.

Another useful type is `Tuple` which represents tuples of elements and convey their arity.

```
(1, 2) :: (Num, Num)
(True, True, False) :: (Bool, Bool, Bool)
(True, 1, 'a', "one") :: (Bool, Int, Char, String)

('a', (False, 'b')) :: (Char, (Bool, Char))
```

```
(['a', 'b'], ('a', [True, False])) :: ([Char], (Char, [Bool]))
```

2.2.2 Functions

Normal functions

Inside Haskell functions' types are represented in order of what types they take as input and what type they produce as output. So we have

```
not :: Bool -> Bool
not True = False
not False = True

add :: (Int, Int) -> Int
add (x, y) = x + y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Note that functions can be partials and so may not work on all inputs. For example we can take the following code

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

The error tells us that `head` cannot work on empty lists.

Curried functions

Generally, while writing Haskell code, we prefer writing

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

as

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

Using this signature we defined a so-called **curried** function, which takes one parameter at the time.
To be more clear, we can see that

```
Int -> Int -> Int -> Int = Int -> (Int -> (Int -> Int))
```

And

```
mult x y z = ((mult x) y) z
```

With this signature we can see that we take one parameter at the time and pass it to the function that is returned by the previous call.

Polymorphic functions

Inside Haskell we can define functions that are **polymorphic**, and so work on all types. An example can be the function

```
length :: [a] -> Int
```

which takes a list of any type and returns its length.
Other polymorphic functions are

```
id :: a -> a
head :: [a] -> a
take :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
```

Overloaded types

As usual inside other languages, `+`, `*` and other operators (which are functions inside Haskell) are overloaded inside Haskell too.

```
Prelude> :t 1 + 2
Num a => a

Prelude> :t 1.1 + 2.0
Fractional a => a

Prelude> :t (+)
Num a => a -> a -> a

Prelude> :t (*)
Num a -> a -> a
```

It is important to note that all types with a type constraint such as `Num =>` or similar, which denote a more specif type than a polymorphic type, are **overloaded** types.

2.2.3 Type classes

Within Haskell we can define **type classes**, which are collections of types that support some overloaded operations called **methods**.

Haskell in particular has some built-in type classes.

Eq

The type class `Eq` contains all the types whose values can be compared for equality and inequality. Its methods are

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

Examples of types that can be found inside `Eq` are `Bool, Char, Int, String and Float`.
Also types like `[a]` and `(a, b, c)` can be found inside `Eq` as long as `a, b` and `c` are inside `Eq`.

Ord

The type class `Ord` contains all the types that are instances of the equality class `Eq` and, in addition, whose values are totally ordered and therefore possess the following methods

```
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
min :: a -> a -> a
max :: a -> a -> a
```

All basic types are inside `Ord`, and lists and tuples are inside `Ord` too provided that their elements have type that is inside `Ord`.
Note that lists, strings and tuples are ordered lexicographically.

Show

The type class `Show` contains all the types whose values can be converted into strings in order to be printed. Its main method is

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

All basic types derive `Show`, and so we can write

```
Prelude> show True
"True"

Prelude> show 'a'
"'a'"

Prelude> show 34
"34"
```

Read

The type class `Read` represents the dual of `Show` and contains all the types whose values can be converted from a string using the method

```
read :: String -> a
```

As usual, most basic types, lists and tuples derive `Read`, so we can write

```
Prelude> read "False" :: Bool
False

Prelude> read "34" :: Int
34

Prelude> read "[2, 4, 5]" :: [Int]
[2, 4, 5]

Prelude> read "('a', False)" :: (Char, Bool)
('a', False)
```

Note that types are mandatory inside these expressions in order to resolve the type of the result which, otherwise, would not be inferrable due to the too few operations. On the other side, the expression

```
Prelude> not(read "False")
```

does not need any type to be inferred as there is already the function `not` which permits the inferring.

Num

The type class `Num` contains all the numeric types. That is all the types be processed with the following methods

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

Integral

The type class `Integral` contains all the types that are inside `Num` and represent integers and support

```
div :: a -> a -> a
mod :: a -> a -> a
```

Note that we should write

```
div 2 3
```

but often we will prefer using the infix notation to write the more comprehensible

```
2 `div` 3
```

Fractional

The type class `Fractional` contains all the types that represents floating point values which are also inside `Num`. Its methods are

```
(/) :: a -> a -> a
recip :: a -> a
```

Chapter 3

Defining functions

Inside Haskell there are a lot of ways to define new functions. Let's analyze and see how we can use each one of them, starting from the easiest one.

3.1 Composition

The first way to define a function in Haskell is by using the composition of older functions. For example we can define a function that tells if a number is even or not by using the `mod` function present inside prelude

```
even :: Integral a => a -> Bool
even n = n ‘mod‘ 2 ==0
```

Or we can also use `take` and `drop` to create a function that splits a list at the n-th value

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

This method is very easy and quick in order to define helper functions, but cannot lead to building very complex functions.

3.2 Conditional

3.2.1 Classic conditional

The second method used to create functions inside Haskell if by using conditionals.
Using this method we can create functions that behave differently based on the input that is given to them.

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n

signum :: Int -> Int
signum n = if n < 0 then -1 else
                        if n == 0 then 0 else 1
```

Note that when using the `if C then T else E` expression, we must ensure that the types of `T` and `E` are the same, in order to avoid type errors. This also means that we **cannot have a dangling else branch**.

3.2.2 Guarded equations

Another way to define functions based on their input is by using **guarded equations**, which are very similar to classic conditionals.

```
abs :: Int -> Int
abs n | n >= 0 = n
          | otherwise = -n

signum :: Int -> Int
signum n | n < 0 = -1
                 | n == 0 = 0
```

```
                    | otherwise = 1
```

We can clearly see that this method makes the functions easier to read when there are more than two choices from which to select.

3.3 Pattern matching

The most common way to define a function in Haskell is by using **pattern matching** which consists in defining a function by the mean of what values its parameters can assume.

```
not :: Bool -> Bool
not False = True
not True = False

(&&) :: Bool -> Bool -> Bool
True && True = True
True && False = False
False && False = False
False && False = False
```

Please note that the patterns defined are evaluated in the same order that they are written, from the top to the bottom. The first pattern that is satisfied determines the output of the function.

If we look at the definition of `&&` given above, we can see that it can be simplified writing

```
(&&) :: Bool -> Bool -> Bool
True && b = b
False && _ = False
```

or

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_ && _ = False
```

We use the special symbol `_` to denote parameters that we won't use and can assume any value. Using this the compiler will perform some optimization in order to run the code faster.

3.3.1 Tuple patterns

As with simple parameters, pattern matching is possible also with tuples.

```
fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y
```

3.3.2 List patterns

Like tuples, patterns can be applied to lists too. The most common pattern that is used while defining functions that work on lists is the pattern that acts when the list is empty and represent thus the base case of a possible recursion.

```
sum :: [Num] -> Num
sum [ ] = 0
sum (x:xs) = x + sum xs
```

Note that with `x:xs` we represent a list that has **at least one value**, as lists can be seen as one value concatenated with a (possible empty) list.

3.4 Lambda expression

Another way to define a function using Haskell is by using **lambda expression**.

```
add :: Int -> Int -> Int
add = \x -> (\y -> x + y)

const :: a -> b -> a
const x = \_ -> x
```

From the signature we can easily see that lambda expressions express naturally currified functions.

Lambdas are often used to create anonymous functions (functions with no name) that are used locally

```
map :: (a -> b) -> [a] -> [b]
map _ [ ] = [ ]
map f (x:xs) = f x : map f xs

odds n :: Int -> [Int]
odds n = map (\x -> (x * 2) + 1) [0..n-1]
```

Chapter 4

List comprehension, recursive functions and higher order functions

4.1 List comprehension

List comprehension is a simple way provided by Haskell to create list using a generator.
Let's see an example

```
Prelude> [ x^2 | x <- [1..5] ]
[1, 4, 9, 16, 25
```

The generator used is `x <- [1..5]` which creates a list containing the values from 1 (inclusive) to 5 (inclusive). The list comprehension than acts as a loop, iterating over all the elements and applying the function which is written on the left side of `|`, obtaining the final list.

Note that when there are multiple generators they acts as nested loops, with the first one written acting as the enclosing loop and the second one as the nested loop.

```
Prelude> [ (x, y) | x <- [1, 2, 3], y <- [4, 5] ]
[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5) ]

Prelude> [ (x, y) | y <- [4, 5], x <- [1, 2, 3] ]
[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]
```

It is also possible to use a first generator to determine the values that a second generator should generate.

```
Prelude> [ (x, y) | x <- [1..3], y <- [x..3] ]
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Also, list comprehension can use guards in order to determine if a value should be put inside the list that is being generated or not.

```
-- | Gets all the factors of a given number
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]

-- | Tells if a number is a prime number or not
prime :: Int -> Bool
prime n = factors n == [1, n]

-- | Generates the first n prime numbers
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

4.2 Recursive functions

Recursive functions are one of the most powerful tools that Haskell provides, but they can cause a lot of trouble during the development too. In order to define a simple and well typed function easily, here are some advices.

1. Define the type of the function
2. Enumerate the cases
3. Define the simpler cases
4. Define the remaining cases
5. Generalize and simplify

Let's see how we can apply these steps while writing `drop`, a function which takes a list as an input and drops the first `n` elements returning the remaining ones.

1. Define the type
   ```
   drop ::  Int -> [a] -> [a]
   ```
2. Enumerate the cases
   ```
   drop 0 [] =
   drop 0 (x:xs) =
   drop n [] =
   drop n (x:xs) =
   ```
3. Define the simple cases
   ```
   drop 0 [] = []
   drop 0 (x:xs) = (x:xs)
   drop n [] = []
   ```
4. Define the remaining cases
   ```
   drop n (x:xs) = drop (n-1) xs
   ```
5. Simplify and generalize
   ```
   drop ::  Integral b => b -> [a] -> [a]
   drop 0 xs = xs
   drop _ [] = []
   drop n (_:xs) = drop (n-1) xs
   ```

4.3 Higher order functions

Higher order functions are functions that return functions as a result. This looks obvious when referring to currying, but with higher order functions we can define also functions that take functions as parameters.

```
twice :: (a -> a) -> a -> a
twice f = f . f

Prelude> twice (*2) 3
12

Prelude> twice reverse [1, 2, 3]
[1, 2, 3]
```

Higher order functions are useful when defining maps, functions that take as input a function and a list to which elements apply the function.

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ fx | x <- xs ]

Prelude> map (+1) [1, 2, 3]
[2, 3, 4]

Prelude> map reverse ["abc", "def"]
["cba", "fed"]
```

Using nested maps it is also possible to work on nested lists

```
Prelude> map (map (+1)) [[1, 2, 3], [4, 5]]
= {applying outer map}
[map (+1) [1, 2, 3], map (+1) [4, 5]]
= {applying inner maps}
[[2, 3, 4], [5, 6]]
```

Also, with higher order functions it is possible to define filters.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

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

Chapter 5

Declaring new types and classes

5.1 Declaring new types

Inside Haskell there are three ways to define a new type.

1. `type`: used when we want to define a so-called **transparent** type, which is a new name for an already existing type
2. `data`, used to define an opaque user defined type
3. `newtype`, used to define especially simple opaque types that allows a more efficient implementation

5.1.1 Type

The simplest way to create a new type is to define another name of an already existing type. This is done using the `type` keyword.

```
type String = [Char]
type Pos = (Int, Int)
type Transer = Pos -> Pos
```

Note that when using `type` we cannot define a recursive tree, and so the current definition is **not allowed**.

```
type Tree = (Int, [Tree])
```

5.1.2 Data

The `data` keyword is used to define whole new types by the meanings of their constructor.
Inside Prelude we can find the following definition

```
data Bool = False | True
```

where `False` and `True` are two constructors for values of type `Bool`.
Note that the order by which the constructors are defined determines the order of the values themselves. So, in this case, the assertion `False < True` would be correct.

An example of a new type definition could be the following

```
data Shape = Circle Float | Rect Float Float
```

were we define `Shape` values to be built either by using the single value constructor `Circle` (which asks for a radius value) or the double value constructor `Rect` (which asks for an height and length values).
Note that the type of `Rect` and `Circle` represents a value, and when we apply the correct values either to `Rect` or `Circle` we get the application itself.

```
Prelude> :t Rect
Rect :: Float -> Float -> Shape

Prelude> Rect 2.3 1.4
Rect 2.3 1.4
```

Once we have defined a new type, we can define also functions to work with values of that type, or utility functions to create new values of that type.

```
square :: Float -> Shape
square x = Rect x x

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Inside Prelude there is a special type, `Maybe`, which is used in order to represent a success or a failure behavior during functions executions.

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

In this case the empty constructor `Nothing` means that there was a failure, while `Just a` represents a success with value `a`.

5.1.3 Newtype

When a type definition involves a single constructor with only one parameter, then it is possible to define it using `newtype` to let the compiler implement it in a more efficient way.

```
newtype Nat = N Int
```

5.1.4 Recursive types

Types defined using `data` and `newtype` can be recursive, while the ones defined using `type` cannot.

```
data Nat = Zero | Succ Nat

nat2int :: Nat -> Int
nat2Int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ(int2nat (n-1))

add :: Nat -> Nat -> Nat
add m n = int2nat(nat2int n + nat2int m)
```

In order to define a binary tree type, when must use `data` because we need to provide a constructor either for the leafs and the nodes of the tree.

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

flatten :: Tree a -> [a]
flatten (Leaf a) = [a]
flatten (Node left x right) = flatten left ++ [x] ++ flatten right
```

5.2 Declaring new classes

5.2.1 Creating a new class

Inside Haskell we can define a new class using the class mechanism.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

Using this definition we are requiring any type that wants to be inside `Eq` to support the methods == and /=, but it is necessary to only define == as /= is defined using == itself.

```
instance Eq Bool where
  False == False = True
  True == True = True
  _ == _ = False
```

It is important to know that only types define using `data` and `newtype` can be made into instances of classes, as an user cannot modify an already existing type using `type`.
Also, as it is inside other programming languages, default method definitions (such as `/=` inside `Eq`) can be overridden inside instances.

5.2.2 Extending a class

Apart from creating a new class, inside Haskell we can extend already existing classes in order to define new ones.

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max :: a -> a -> a

  min x y | x <= y = x
          | otherwise = y

  max x y | x <= y = y
          | otherwise = x
```

Using this definition by extension, if a type is inside `Eq` and wants to be also `Ord`, it will have to implement four methods (`<, <=, >, >=`) and will get two methods for free due to their default implementation (`min` and `max`).

```
instance Ord Bool where
  False < True = True
  _ < _ = False
  b <= c = (b < c) || (b == c)
  b > c = c < b
  b >= c = c <= b
```

If we want to create a new type and make it part of a class, than all we have to do is use the `deriving` keyword.

```
data Bool = False | True
                    deriving (Eq, Ord, Show, Read)
```

Note that, as seen before, the order `False < True` is determined by the constructors declaration order.

Chapter 6

Interactive programming and I/O

6.1 Introduction to I/O in Haskell

So far we've seen only batch programs in Haskell. Inside them the input is given together with the program that executes in order to print the result.
Most of the programs, however, are interactive and ask for input in order to return an output possibly several times during the execution. Since Haskell is functional we want to see I/O actions as functions too, in particular of type `IO`.

```
type IO = World -> World
```

This type means that every I/O action will take as input the current state of the execution (called `World`) and return a new state.
A part from the change of the state of the execution, I/O actions may however return a value too. So we can change the definition and write the following.

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

By this definition, we identify `IO Char` which returns a `Char` and `IO ()` which performs pure side effects.
In order to describe an input we can then use currying to define `Char -> IO ()`, which takes a `Char` and performs side effects.

Now that we have defined an IO action in terms of functions, it is natural to ask ourselves: "What is `World`?".
The reality is that we don't (or, better, do not want to) know what is. It is sufficient for us to know that in reality `IO` is a built-in type whose details are hidden.

6.2 Basic I/O actions

Some basic I/O actions that can be composed in order to create more sophisticated interactive programs are the followings.

```
getChar :: IO Char
putChar :: Char -> IO ()
return :: a -> IO a
```

These actions are all built in into the GHC system, and their implementations won't be object of this course.
Note, however, that `return` transforms any expression into an IO action that delivers that expression.

As the type `IO a` is a monad, we can use a special notation to compose I/O actions.

```
do v1 <- a1
   v2 <- a2
      ...
   vn <- an
   return (f v1 v2 ... vn)
```

Where each `v <- a` is a generator.
We can also use `a` alone without `v <-` if we don't care about `v` itself.

To make the things more clear, let's see an action that reads three characters and returns the first and third ones.

```
act :: IO (Char, Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x, y)
```

Note that omitting `return` would result in a type error.

6.3 Derived primitives

Let's see how we can write useful functions using the ones already present inside Prelude.

```
-- | Gets a line as a string
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then return []
             else do xs <- getLine
                     return (x:xs)

-- | Prints a string
putString :: String -> IO()
putString (x:xs) = do putChar x
                      putString xs

-- | Prints a string as a line
putStringLine :: String -> IO ()
putStringLine = do putString xs
                   putChar '\n'
```

Let's see now an I/O action which prompts for a string and then displays its length.

```
stringLen :: IO()
stringLen = do putString "Enter a string:"
               xs <- getLine
               putString "The string has "
               putString (show (length xs))
               putString " characters"
```

Chapter 7

Functors

7.1 Introduction

So far we've encountered a lot of the type classes that are present inside Prelude. We've seem how `Eq` represents the types that can be equated and `Ord` the ones that can be ordered. We've used `Show` to classify the types that can be displayed and `Read` the ones that can be read.
An now, we are talking about the `Functor` type class, which is basically for things that can be mapped over. The first reaction when listening to the keyword "map" is probably thinking about lists, and that is perfect because the list type of part of the `Functor` class.

Let's see now how `Functor` is implemented.

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

We see that it defines one function, `fmap`, and doesn't provide any default implementation for it. We see that `fmap` takes a function from one type to another and a functor applied with one type, and returns a functor applied with another type.

7.2 Examples

7.2.1 Lists

In order to make things more clear, let's take a step back and see a similar type, the one of `map`.

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

Interesting! It takes a function from one type to another and a list of one type, and returns a list of another type. This is a perfect example of `Functor`. In fact, `map` is just a `fmap` that works only on lists and is defined as follows.

```
instance Functor [] where
  fmap = map
```

Notice that there is no need to write `instnace Functor [a] where`, because from `fmap :: (a -> b) -> f a -> f b` we see that `f` has to be a type constructor that takes one type. `[a]` is already a concrete type (of a list with any type inside it), while `[]` is a type constructor that takes one type and can produce types such as `[Int], [String]` or even `[[String]]`.

Since for lists `fmap` is just `map`, we get the same results when using them on lists.

```
Prelude> fmap (*2) [1..3]
[2, 4, 6]

Prelude> map (*2) [1..3]
[2, 4, 6]
```

7.2.2 Maybe

Similar to `[]`, also `Maybe` is a `Functor`.

```
instance Functor Maybe where
  fmap f (Just x) = Just(f x)
  fmap f Nothing = Nothing
```

7.2.3 Tree

Another type that can be mapped over is `Tree a`.

```
instance Functor Tree where
  fmap f (Leaf a) = Leaf(f a)
  fmap f (Node left x right) = Node (fmap f left) (fmap x) (Node fmap f right
     )
```

7.3 Wrapping up

In a more general way, a `Functor T a` is a container that contains values of type `a` and `fmap f` applies `f` to each value inside the container.
We can see that `IO` is an instance of `Functor` too.

```
instance Functor IO where
  fmap f mx = do x <- mx
                 return (g x)
```

The usage of `fmap` is that it can process the values of any container that is a `Functor`, and this generalization propagates to functions defined by the mean of `fmap` itself.

```
inc = fmap (+1)

Prelude> inc (Just 1)
Just 2

Prelude> inc [1, 2, 3]
[2, 3, 4]

Prelude> inc (Node (Leaf 1) 0 (Leaf 2))
Node (Leaf 2) 1 (Leaf 3)
```

7.4 Functor laws

In order to define a well behaved `Functor` we need to make it respect some laws.

1. `fmap id = id`
   That is, `fmap` needs to preserve the identify.

2. `fmap (f . g) = fmap g . fmap h`
   That is, `fmap` needs to preserve the function composition.

These rules, together with the type of `fmap`, assure that `fmap` is a mapper that does not reorder the values inside the container which is applied to.

7.5 Interesting fact

For any parameterized type inside Haskell there is `at most one` function `fmap` that satisfies the laws. So, if we can make a parameterized type into a `Functor`, then `fmap` is unique.

Chapter 8

Applicatives

8.1 Introduction

As we've seen before, functions in Haskell are curried by default, which means that a functions that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on. If a function is of type `a -> b -> c`, we usually say that it takes two parameters and returns a `c`, but actually if takes an `a` and returns a functions `b -> c`. That's why we can call a function as `f x y` or as `(f x) y`. This mechanism is what enables us to partially apply functions just by calling them with too few parameters, which results in functions that we can then pass on to other functions.

So far,when we were mapping functions over functors, we usually mapped functions that take only one parameter. But what happens when we map a function like `*`, which takes two parameters, over a functor? Let's take a couple of concrete examples of this. If we have `Just 3` and we do `fmap (*) (Just 3)`, what do we get? From the instance implementation of `Maybe` for `Functor`, we know that if it's a `Just` *`something`* value, it will apply the function to the *`something`* inside the `Just`. Therefore, doing `fmap (*) (Just 3)` results in `Just ((*) 3)`, which can also be written as `Just (* 3)`. So, we got a function wrapped inside `Just`!

So, we see that by mapping "multi-parameter" functions over functors we get functors that contain functions inside them. But what can we do with them? Well for one, we can map functions that take there functions as parameters over them, because whatever is inside a functor will be given to the function that we're mapping over it as parameter.

```
Prelude> let a = fmap (*) [1, 2, 3, 4]
Prelude> :t a
a :: [Integer -> Integer]
Prelude> fmap (\f -> f 9) a
[9, 18, 27, 36]
```

But what if we have a functor value of `Just (* 3)` and a functor value of `Just 5` and we want to take out the function from `Just (3 *)` and map it over `Just 5`? With normal functors, we're out of luck, because all they support it just mapping normal functions over existing functors. Even when we mapped `f -> f 9` over a functor that contained functions inside it, we were just mapping a normal function over it. But we can't map a functor that's inside a functor over another functor with what `fmap` offers us. We could pattern-match against the `Just` construct to get the function out of it and then map it over `Just 5`, but we're looking for a more general and abstract way of doing that which works across functors.

This is were the `Applicative` type class comes in. It lies inside the `Control.Applicative` module inside Prelude and it defines two methods, `pure and <*>`. It doesn't provide a default implementation for any of them, so we have to define them both if we want something to be an applicative functor. The class is define like so.

```
class (Functor f) => Applicative f where
```

```
  pure :: a -> f a
  (<*>) :: f(a -> b) -> f a -> f b
```

This simple three line class definition tells us a lot! Let's start at the first line. It starts the definition of `Applicative` class and it also introduces a class constraint. It says that if we want to make a type constructor part of the `Applicative` type class, it has to be in `Functor` first. That's why if we know that if a type constructor is part of the `Applicative` type class, it's also in `Functor`, so we can use `fmap` on it.

The first method it define is called `pure`. It's type declaration is `pure :: a -> f a`. `f` plays the role of out applicative functor instance here. Because Haskell has a very good type system and because everything a function can do is take some parameters and return some value, we can tell a lot from a type declaration and this is no exception. `pure` should take a value of any type and return an applicative functor with that value inside it. So, we take a value and we wrap it in an applicative functor that has that values as the result inside it.

A better way of thinking about `pure` would be to say that it takes a value and puts it in some sort of default (or *pure*) context - a minimal context that still yields that value.

The `<*>` function is really interesting. IT has a type declaration of `f (a -> b) -> f a -> f b`. It is very similar to `fmap :: (a -> b) -> f a -> f b`. It's a sort of a beefed up `fmap`. Whereas `fmap` takes a function and a functor and applies the function inside the functor, `<*>` takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one.

8.2 Examples

8.2.1 Maybe

Let's take a loot at the `Applicative` instance implementation for `Maybe`.

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

Again, from the class definition we see that `f` that plays the role of the applicative functor should take one concrete type as a parameter, si we write `instance Application Maybe where` instead of writing `instance Applicative (Maybe a) where`.

First of,, `pure`. We said earlier that it's supposed to take something and wrap it in an applicative functor. We wrote `pure = Just`, because value constructors like `Just` are normal functions. We could have also written `pure x = Just x`.

Next up, we have the definition for `<*>`. We can't extract a function out of a `Nothing`, because it has no function inside it. So we say that if we try to extract a function from a `Nothing`, the result is a `Nothing`. If you look at the class definition for `Applicative`, you'll see that there's a `Functor` class constraint, which means that we can assume that both `<*>`'s parameters are functors. If the first parameter is not a `Nothing`, but a `Just` with some function inside it, we say that we then want to map that function over the second parameter. This also takes care of the case where the second parameter is `Nothing`, because doing `fmap` with any function over a `Nothing` will return a `Nothing`.

```
Prelude> pure (+) <*> Just 3 <*> Just 5
```

```
Just 8

Prelude> pure (+) <*> Just 3 <*> Nothing
Nothing

Prelude> pure (+) Nothing <*> just 5
Nothing
```

Note that `<*>` if left-associative, which means that `pure (+) <*> Just 3 <*> Just 5` is the same as `(pure (+) Just 3) <*> Just 5`.

8.2.2 Lists

One instance of `Applicative` is `[]`.

```
instance Applicative [] where
  pure x = [x]
  functions <*> values = [f x | f <- functions, x <- values]
```

So we can use this definition as follows in order to map a list of functions each one over every single value of a list of values.

```
Prelude> [(*0), (+100), (^2)] <*> [1, 2, 3]
[0, 0, 0, 101, 102, 103, 1, 4, 9]
```

8.2.3 IO

Another instance of `Applicative` that we've encountered is `IO`. This is how the instance is implemented.

```
instance Applicative IO where
  pure = return
  a <*> b = do
    f <- a
    x <- b
    return (f x)
```

Since `pure` is all about putting a value in a minimal context that sill holds it as its result, it makes sense that `pure` is just `return`, because `return` does exactly that; it makes an I/O action that doesn't do anything, it just yields some value as its result, but it doesn't really do any I/O operations like printing to the terminal or reading from a file.

Chapter 9

Monads

9.1 Introduction

When we started off with functors, we saw that it's possible to map functions over various data types. We saw that for this this purpose, the `Functor` type class was introduces and it had us asking the question: when we have a function of type `a -> b` and some data type `f a`, how do we map that function over the data type to end up with `f -> b`? We was how to map something over a `Maybe a`, a list `[a]` and `IO a` etc. To answer this question of how to map a function over some data type, all we had to do was look at the type of `fmap`.

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

And then make it work for our data type by writing the appropriate `Functor` instance.

Then we saw a possible improvement of functors and said, hey, what if that function `a -> b` is already wrapped inside a functor value? Like, that if we have `Just (*3)`, how do we apply that to `Just 5`? What is we do not want to apply it to `Just 5` but to a `Nothing` instead? Or if we have `[(*2), (+4)]`, how do we apply that to `[1, 2, 3]`? How would that work even? For this, the `Applicative` type class was introduced, in which we wanted the answer to the following type.

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

We also saw that we cant take a normal value and wrap it inside a data type. For instance, we can take a `1` and wrap it so that it becomes a `Just 1`, Or we can make it to a `[1]`. Or an I/O Action that does nothing just yields `1`. The function that does this is called `pure`.

Monads are a natural extension of applicative functors and with them we're concerned with this: if you have a value with a context, `m a`, how do you apply to it a function that takes a normal `a` and returns a value with a context? That is, how do you apply a function of type `a -> m b` to a value of type `m a`? So, essentially, we will want the following function.

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

If we have a fancy value and a function that takes a normal value but returns a fancy value, how do we feed that fancy value into the function? This is the main question that we will concern ourselves when dealing with monads. We write `m a` instead of `f a` because the `m` stands for `Monad`, but monads are just applicative functions that support `»=`. The `»=` function is pronounced as *bind.*

Now that we have a vague idea of what monads are about, let's see if we can make that idea a bit less vague, considering the type `Maybe`.

A value of type `Maybe a` represents a value of type `a` with the context of possible failure attached. A value `Just "dharma"` means that the string `"dharma"` is there whereas a value of `Nothing` represents its absence, or if you look at the string as the result of a computation, it means that the computation has failed.

Let's then think about how we would do »= for `Maybe`. Like we said, »= takes a monadic value and a function that takes a normal value and returns a monadic value, and manages to apply that function to the monadic value. How does it do that, if the function takes a normal value? Well, to do that it has to take into account the context of that monadic value.

In this case, »= would take a **Maybe a** value and a function of type `a -> Maybe b` and somehow apply the function to the `Maybe a`. To figure out how it does that, we can use the intuition that we have from `Maybe` being an applicative functor. Let's say that we have a function
`x -> Just (x + 1)`. It takes a number, adds `1` to it and wraps it in a `Just`.

```
Prelude> (\x -> Just(x + 1)) 1
Just 2

Prelude (\x -> Just(x + 1)) 100
Just 101
```

If we feed `1` it evaluates to `Just 2`. If we give it the number `100` the results is `Just 101`. Now, how do we feed a `Maybe` value to this function? If we think about how `Maybe` acts as an applicative functor, answering this is pretty easy. If we feed it a `Just` value, take what's inside the `Just` and apply the function to it. If we give it a `Nothing`, then we're left with a function but `Nothing` to apply to it. In this case, let's just do what we did before and say the result is `Nothing`.

Instead of calling it »=, let's call it `applyMaybe` for now.

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

Prelude> Just 2 ‘applyMaybe‘ \x -> Just(x + 1)
Just 4
```

So, it looks like that for `Maybe` we've figured out how to take a fancy value and feed to to a function that takes a normal values and returns a fancy one.

9.2 The Monad type class

Just like functors have the `Functor` type class and applicative functors have the `Applicative` type class, monads come with their own type class: `Monad`.

```
class Monad m where
  return :: a -> m a

  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  x >> y = x >> \_ -> y

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

Let's start with the first line. It says `class Monad m where`. But shouldn't it be something like `class (Applicative m) => Monad m where` so that a type has to be an applicative functor first before it can be made a monad? It really should, but when Haskell was born it hadn't occurred to people that applicative functors are a good fit for Haskell so they weren't in there. But rest assured, every monad is an applicative functor.

The first function that the Monad type class defines is return. It's the same as pure, only wit ha different name. It takes a value and puts it in a minimal default context that sill holds that value.

The next function is »=, or bind. It's like the function application, only instead of taking a normal value and feeding it to a normal function, it takes a monadic value and feeds it to a function that takes a normal value but returns a monadic value.

Next up, we have »=. We won't pay too much attention to it for now because it comes with a default implementation and we pretty never implement it when making Monad instances.

The final function of the Monad type class is fail. We never use it explicitly in our code. Instead, it's used by Haskell to enable failure in a special syntactic construct for monads. We don't need to concern ourselves with fail too much for now.

9.3 Examples

9.3.1 Maybe

Now that we know that the Monad type class looks like, let's take a look at how Maybe is an instance of Monad.

```
instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
  fail _ = Nothing
```

return is the same as pure. We do what we did in the Applicative type class and wrap it in a Just.

The usefulness of the »= function is that it lets you contact sequentially different operations, and propagates the error.

```
Prelude> Just 1 >>= Just 2 >>= Nothing >>= Just 3
Nothing
```

Because of how »= is define, the first Nothing value that is encountered will result in it being propagated, and so all the computation will result in Nothing.

9.4 The do notation

Monads in Haskell are so useful that they got their own special syntax called do notation. We've already encountered do notation when we were doing I/O and there we said it was used to glue together I/O actions into one. The interesting thing is that the do notation can be used for every monadic type. Its principle is still the same: gluing together monadic values in sequence.

Consider the following script.

```
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))
```

The result of this script is `Just("3!")`, but we can clearly see that the syntax is pretty complex. We have to lambdas just to perform simple operations.
With the `do` notation we can remove the lambdas to obtain a cleaner code.

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

In a `do` expression, every single line is a monadic value. To inspect its result, we use `<-`. If we have a `Maybe String` and we bind it with `<-` to a a variable, that variable will be a `String`, just like when we used `>>=` to feed monadic values to lambdas. The last monadic value in a `do` expression, like `Just(show x ++ x)` here, can't be used with `<-` to bind its result, because that wouldn't make sense if we translated the `do` expression back to a chain of `>>=` applications. Rather, its result is the result of the whole glued up monadic value, taking into account the possible failure of any of the previous ones (which is in fact passes from one to the next monadic value).

Chapter 10

Lazy evaluation

AS we've seen since now, inside Haskell the computation is the application of function to given arguments. As everything can be considered a function, there are cases where some functions can be evaluated in different ways.
Let's consider the following function.

```
int :: Int -> Int
inc n = n + 1
```

It's application `int (2 * 3)` can be evaluated either in the following order

```
inc (2 * 3)
= applying inc
(2 * 3) + 1
= applying *
6 + 1
= applying +
= 7
```

or the following one

```
inc (2 * 3)
= applying *
inc (6)
= applying inc
6 + 1
= applying +
7
```

Fortunately, inside Haskell any two different ways of evaluating the same expression will always produce the same final value, provided that they both terminate.
This is possible due to the **absence of side effects** inside functions characteristic of Haskell. Inside common imperative languages this is not possible. Let's consider the function `n + (n = 1)` and its following ways of evaluation with a starting value of `n = 0`.

```
n + (n = 1)
= applying n
0 + (n = 1)
= applying =
0 + 1
= applying +
1

n + (n = 1)
= applying =
n + (1)
= applying n
1 + 1
= applying +
2
```

As we can see the presence of side effects can lead to different results based on the evaluation strategies.

10.1 Evaluation strategies

Before defining the different evaluation strategies that can be found inside Haskell, let's define a base concept. An expression that has the form of a function applied to one or more arguments is called a **reducible expression** (**redex**).
For example, considering the following function definition

```haskell
mult :: (Int, Int) -> Int
mult (x, y) = x * y
```

then the expression `mult(1 + 2, 2 + 3)` contains three redexes. The first is `1 + 2`, which reduced will lead to `mult(3, 2 + 3)`. The second one is `2 + 3`, which would lead to `mult(1 + 2, 5)` and the third one is `mult(1 + 2, 2 + 3)` which would lead to `(1 + 2) * (2 + 3)`.
Even if all of the three ways of reducing wouldn't produce a different value, it is important to determine which one of them should be reduced first, which one second and which one third.

10.1.1 Innermost evaluation

The first evaluation strategy is called the **innermost evaluation**, and consist into picking the redex which does not contain other redexes. When there are more than one redexes that do not contain other redexes, than the evaluation order must be from the left to the right.

Using the example above, the evaluation path would then result into the following.

```
mult(1 + 2, 2 + 3)
= applying the left-most +
mult(3, 2 + 3)
= applying the +
mult(3, 5)
= applying mult
3 * 5
= applying *
15
```

It's important to note that while using the innermost evaluation, arguments are passed to functions **by value**. They are in fact completely computed before being passed inside the function body.

10.1.2 Outermost evaluation

Dual to the innermost evaluation it's the **outermost evaluation** strategy, inside which the redexes are computed starting from the ones that are not contained inside other redexes and from the left to the right.

Following the `mult` example, we would obtain the following evaluation path.

```
mult(1 + 2, 2 + 3)
= applying mult
(1 + 2) * (2 + 3)
= applying the left +
3 * (2 + 3)
= applying +
3 * 5
= applying *
15
```

Opposed to what happens inside the innermost evaluation strategy, while using the outermost evaluation arguments are passed to function bodies not evaluated. This particular strategy is called **by name**.

10.2 Strict functions

If we look closer to the examples presented, we can see that some functions (such as `+` and `*`) require their arguments to be fully evaluated before being called, even during the outermost evaluation strategy. Such functions are called **strict** and cannot work without fully evaluated arguments.

10.3 Lambda expressions

Let's take a look at what happens when dealing with lambda expression.

```
mult :: Int -> Int -> Int
mult x = \y -> x * y
```

The first evaluation strategy we consider is the innermost evaluation.

```
mult(1 + 2, 2 +3)
= applying left +
mult(3, 2 + 3)
= applying mult
(\y -> 3 * y)(2 + 3)
= applying +
(\y -> 3 * y)(5)
= applying lambda
3 * 5
= applying *
15
```

As we can see, the arguments are passed **by value** one at the time. If we would use the outermost evaluation strategy we would have to remember that **the selection of redexed withing the body of a lambda expression is forbidden**. That said, we cannot reduce the body of a lambda, as their are see as black boxes.
For example, the expression (
`x -> 1 + 2)` is considered as fully evaluated, and we cannot manipulate the body `1 + 2` in order to reduce it deleting the lambda.

10.4 Termination

Now that we've seen the different evaluation strategies, let's see how they can yield to different results when taking into consideration the computation of non-terminating functions.

Let's consider the following functions

```
fst :: (Int, Int) -> Int
fst (x, _) = x

inf :: Int
int = 1 + inf
```

and the expression

```
fst(0, inf)
```

It's easy to note that, if we would apply innermost evaluation, the computation would not terminate, because we would have to fully compute the `inf` function before passing its value to `fst`. But, as soon as we start to try to compute `inf`, this does not terminate due to an infinite recursion.
Looking closer at the definition of `fst`, we notice that we should not compute `inf` at all, as `fst` itself doesn't even consider the value of the second value of the pair to return the first one.
This observation is what makes outermost evaluation successful inside this example, as it lets execute `fst` first, which returns simply `0` and ends the computation.

The things that we saw explicitly inside this example can be expressed inside a theorem stating that **is there is any evaluation sequence that terminates for a given expression, then call by name evaluation will also terminate for that expression and with the same final result**.
On the other side, we cannot state anything about call by value, as it may or may not terminate.

10.5 Number of reductions

Let's now consider the following function

```
square :: Int -> Int
square n = n * n
```

Trying to evaluate `square(1 + 2)` using call by value will result in the following evaluation

```
square(1 + 2)
= applying +
square 3
= applying square
3 * 3
= applying *
9
```

while using call by value would result into the following one

```
square(1 + 2)
= applying square
(1 + 2) * (1 + 2)
= applying left +
3 * (1 + 2)
= applying +
3 * 3
= applying *
9
```

As we can see, the call by name evaluation is longer than the call by value, just because the expression `1 + 2` is computed two times.
This problem can be fixed by using **pointers** to indicate the sharing of an expression during the evaluation.
Using pointers the compiler is able to evaluate the expression `1 + 2` only one time, and then replace its value where needed, cutting down one step of evaluation and making it as long as the call by value evaluation.

The usage of call by name evaluation and memory pointers is what determines the ability of perform **lazy evaluation** inside Haskell.

Other than avoid the computation of same expressions twice or more times, lazy evaluation also permit the usage of infinite structures such as lists.
Let's consider the following function.

```
ones :: [Int]
ones = 1 : ones
```

The computation of `ones` does not terminate due to its infinite recursion, but it also avoid the expression `head ones` to terminate too.

```
head ones
= head (1 : ones)
= head (1 : (1 : ones))
= head (1 : (1 : (1 : ones)))
...
```

However, using the lazy evaluation, we are able to solve this problem.

```
head ones
= applying ones
head 1:ones
= applying head
1
```

Due to the fact that lazy evaluation evaluates `ones` only as much as it is strictly necessary in order to produce the result, we are able to end the computation without falling for an infinite recursion.
So, it is important to note that **with lazy evaluation expressions are only evaluated as much as required by the context in which they are used**. And this does not apply only to lists, but also to trees and other potentially-infinite data structures.

10.6 Creating strict functions

As seen above, Prelude includes a set of so-called *stict functions*. These functions require one or more of their arguments to be fully evaluated before executing. But what if we want to define a stric function? How should be do it?
Fortunately for us, Haskell provides the `$!` function, which **forces the top-level evaluation of the argument to which is applied**. This means that if the argument is a basic type (`Int, Bool`, etc) the evaluation stops when it reaches a constant. If it is a pair, then it stops as soon as it has shown that the argument as a value of type `(_, _)`. Finally, if it is a list then it stops as soon as the value is either `[]` or `(_:_)`.

Let's consider the following function as an example.

```
square :: Int -> Int
square x = x * x
```

And let's apply it using `$!` to its parameter.

```
square $! (1 + 2)
= applying +
square $! (3)
= applying $!, it stop because the argument is and Int and is a constant
   value
square 3
= applying square
3 * 3
= applying *
9
```