# FP(H) — Lab Sheet 1

Jeremy Singer

Mon 30 Sep 2019

## 1 Learning Objectives

This is the first Haskell lab, so we want you to become familiar with the Haskell tools installed on the lab machines. You should be able to execute ghci from a bash prompt. Once you can do this, you should work on the exercises below, with the aim of:

- recognizing and producing a broad range of Haskell *expressions*
- understanding how simple Haskell expressions are *evaluated*
- comparing small Haskell fragments with programs in imperative languages
- design Haskell *functions*, including those that feature *pattern matching*
- understand the Haskell *list* data structure

## 2 Simple Puzzles

### 2.1 Doubling up

Write a function `double :: Int -> Int` which returns two times the value of its input parameter. Can you think of three different ways to write this function definition?

### 2.2 Some like it hot

The *fahrenheit* temperature is the *celsius* temperature multiplied by 9/5, plus 32. Write a function `c2f :: Float -> Float` to convert Celsius to Fahrenheit.

### 2.3 Thirds

You may have encountered the fst and snd functions for pairs. How would you define a function `thd :: (a,b,c)-> c` for triples?

### 2.4 Sort it out

Can you explain in plain English what the following Haskell code does? What is the result of evaluation?

```haskell
let numbers = words "one two three four five six seven eight nine ten"
    backnums = map reverse numbers
in last $ sort backnums
```

## 3 A Longer Exercise: Roman Numerals

Develop a function that takes in a *roman numeral* as input, and returns the corresponding integer value. Assume the Roman numeral is in the range $[0, 400)$. The function should have the following type signature:

```haskell
    romanToInt :: String -> Int
```

**Functional Programming in Haskell – Lecture 2 – Wed 2 Oct 2019**

**My first function**

Functional programming is all about *functions*. In FP, functions are first-class citizens – i.e. a function is a value just like an `Integer` is a value – you can pass it around as a parameter, bind it to a name, etc.

Anonymous functions, or lambdas, are defined with the backslash '\' character

```
\x->x+1
```

This is a lambda abstraction, or an unevaluated function. It's waiting for an argument to be supplied…

```
(\x->x+1) 41
```

This is a lambda application, or a function call. The argument has been supplied, so the function evaluation can take place (insert forward reference to laziness here).

Let's bind this function to a name:

```
inc = \x->x+1
```

Now we can say

```
inc 41
```

which is an equivalent lambda application.

In Haskell source code files, we generally lay out a function definition as follows:

```
-- | say something nice about the input String value, based on its length
compliment :: String -> String
compliment thing =
    let suffix = if length thing `mod` 2 == 0
                        then "evenly excellent"
                        else "oddly awesome"
    in thing ++ " is " ++ suffix
```

Think about
1) how to deal with singular and plural 'thing's – maybe add a Boolean flag argument? This leads on to the notion of currying
2) are there other ways you might define a compliment function with equivalent behaviour?

**Functional Programming in Haskell – Lecture 3 – Mon 7 Oct 2019**

## Recursion

In imperative programming, most of the 'interesting' computation happens inside loops. In functional programming, we don't use loops. Instead, repeated computation takes place via recursive function calls.

## What is recursion?

Recursion occurs when a concept or entity is defined in terms of itself. This kind of self-referential behaviour is common in all of Computing Science[1], but particularly in functional programming.

Let's examine a simple recursive function definition, to compute factorial numbers:

```
fact :: Int->Int
fact 0 = 1
fact n = n * fact (n-1)
```

Notice we need a base case (for input argument 0). Unless there is at least one base case then the function evaluation will diverge, and you end up with an out-of-memory error in the Haskell runtime.

Notice the base case comes first – since it is more specific than the general case. Function patterns are matched from top-to-bottom, so if the top one matches then it takes precedence over lower down patterns.

The recursive case comes second, for `fact.` The function body here performs a small amount of computation (a single multiplication, which is in fact deferred) and a *tail call* to another invocation of `fact.` Notice that the argument to the tail call is *smaller* than the argument originally supplied to this function invocation – this is the guarantee that we will *make progress* in the function evaluation.

## Write some recursive functions

Let's try to define some more recursive functions.

```
sum :: Int->Int
-- ^ sums the first n integers, for input value n

fib :: Int->Int
-- ^ computes the nth Fibonacci number, for input value n


replicate :: Int->[Int]
-- ^ generate a list of n values, each value is n
```

[1] Look up 'recursion' in the K&R C textbook, or search for 'recursion' on Google

## Recursion with an accumulator

Sometimes, we want to accumulate a value as we recurse, for instance:

```
fact2 :: Int->Int->Int
fact2 0 acc = acc
fact2 n acc = fact (n-1) (acc*n)
```

What might be the reasons for this accumulation of data?

Can you rewrite your three functions from the previous page using an accumulator?

## Recursion over lists

It is possible to recurse over a data structure. Consider a list – the base case is the empty list and the recursive case is a list with at least one element. We can use pattern matching on the data structure to specify these different cases.

```
sumList :: [Int]->Int
sumList [] = 0
sumList (x:xs) = x + sumList xs
```

Let's try to define some more recursive functions over lists.

```
prodList :: [Int] -> Int
lengthOfList :: [Int] -> Int
reverseList :: [a] -> [a]
maxInList :: [Int] -> Int
```

**Functional Programming in Haskell – Lecture 5 – Mon 14 Oct 2019**

## Algebraic Data Types

Custom data types can be used to model a domain of values. This allows us to express computation over the domain, and reason about it in a high-level manner. In traditional languages like C or Java, we are used to defining custom data types with `enum` or `struct` style definitions. Let's explore how to create similar data types in Haskell.

## Simple Sum Data Types

A *sum* data type defines a set of alternative values. Perhaps the simplest sum data type is `Bool`, which can have values `True` or `False`.

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

Such sum data types are similar to enumerated types in imperative / OO languages. Here is another example:

```
data Emotion = Happy | Sad | Confused | Angry   deriving (Show,Eq)
```

The deriving clause indicates typeclass behaviour we want to specify for this type – we want to be able to convert Emotions to Strings and compare Emotions for equality.

Once we have defined a sum data type, we can write functions over it:

```
greetPerson Happy = "great to see you!"
greetPerson e = "sorry you are feeling " ++ (show e)
```

## Simple Product Data Types

A product type combines two other values into a compound value. For instance, a 2-d coordinate is a product type.

```
data Coordinate = Coord Int Int
```

Product types may combine values of different types. For instance, a University course code type is a product:

```
data CourseCode = Code String Int
```

Such product data types are similar to structs in C or class definitions in OO languages like Java. It is possible to specify a more complex domain as a sum of products – a little like a set of interface implementations in Java.

```
data Vehicle = Car Int String Fuel
             | Bicycle Int Bool
             | Bus Int String
```

The fields for each Vehicle type mean something different for each type – perhaps the Car needs a registration plate String whereas the Bike only needs a Boolean to indicate whether it is electric or not. We will look at record syntax in later lectures, to help us enrich the data type syntax to make fields more meaningful.

**Pattern Matching with Product Data Types**

We can write functions that pattern match on different values of a product data type, for instance:

```
isGreen :: Vehicle -> Bool
isGreen Car _ _ Electric = True
isGreen Bicycle _ _ = True
isGreen _ = False
```

We can also bind names to matched fields in the pattern match, for instance:

```
isBig :: Vehicle -> Bool
isBig Bus decks _ = if decks > 1 then True else False
isBig _ = False
```

**Custom show functions**

In Haskell, `show` is like `toString` from Java or `str` from Python. It converts values from arbitrary data types into String values for output. If a data type is defined as `deriving Show` then a default show function is provided.

If a data type is declared to be an instance of the Show typeclass then a custom show function can be defined.

```
instance Show Emotion where
    show Happy = "☺"
    show Sad = "☹"
    show Angry = ":-~"
    show Confused = ";-\"
```

**Functional Programming in Haskell – Lecture 6 – Wed 16 Oct 2019**

## Recursive Data Types

In the same way that we can define recursive functions that call themselves, we can also define recursive data types that contain themselves.

```
data MyIntList = Nil | Cons Int MyIntList
```

This is similar to the built-in list data structure, and we can define similar list functions for this type. Note that `MyIntList` can only contain `Int` values – we need to use polymorphism to make the List type more general …

## Polymorphic Types

Type variables (like `a` that we have seen earlier) allow us to have 'holes' in data types that we can fill in later (i.e. bind to concrete types). Such polymorphic types are universally quantified in some way over all types. Polymorphic type expressions describe *families of types* – consider `(a,a)` as a family of pair types – including `(Int, Int)`, `(Bool, Bool)`, etc.

We can define a List like container type that is polymorphic as follows:

```
data MyList a = Nil | Cons a MyIntList
```

Note the use of the type variable `a`. `MyList` is a *type constructor* – we provide a type argument (which is bound totype variable `a`) in order to get a concrete type – e.g. `MyList Int`.

Q: What is the difference between a *type constructor* and a *data constructor* in Haskell?

Q: What is a *kind* in Haskell?

## Binary Trees

While it is interesting to define the List data type, this is actually already defined for us in the Haskell language. Let's look at the binary tree data structure for a further example of a recursive data type.

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

We might also want a typeclass clause `deriving (Show,Eq)`. There are various alternative tree definitions – are values stored at internal nodes or at leaves, or both? We could sketch out some of these alternative definitions…

## Operations on Binary Trees

We can define recursive functions over binary trees by following the structure of the recursive data type in our pattern match. Here is an example function to count the number of elements stored in a tree.

```
treeSize :: Tree a -> Int
treeSize Leaf = 0
treeSize (Node _ left right) = 1 + (treeSize left) + (treeSize right)
```

Consider how you might define other functions to find the sum of Ints in a Tree Int, or the maximum depth of a Tree, or whether a particular value is stored in a tree, or convert the Tree to a list.

Notice the *typeclass constraints* that are induced by various operations – for instance if we want to see if a value is in a tree, we need the type of the value to be an instance of the `Eq` typeclass so we can use the `==` operator. In the type signature, we use the 'fat arrow' `=>` to indicate this typeclass constraint on the polymorphic type variable.

```
treeElem :: Eq a => a -> Tree a -> Bool
```

## Mapping over Binary Trees

The `treeMap` function is particularly interesting. This preserves the structure of the tree but operates on the individual elements contained.

```
treeMap :: (a->b) -> Tree a -> Tree b
```

We will revisit the `treeMap` function when we consider the notion of *functors*, later in the course. Search for *functors in Haskell* if you want to get ahead!

There are actually binary tree packages available for Haskell – e.g. see http://hackage.haskell.org/package/binary-tree

**Functional Programming in Haskell – Lecture 9 – Mon 28 Oct 2019**

## Unit Testing in Haskell

Just like Java features JUnit, there is a standard unit testing framework in Haskell called (unsurprisingly) HUnit. This allows you to specify unit tests in Haskell, group them together and execute them to get a test report. The framework is in a Haskell package called *HUnit* (case sensitive). It is probably installed by default with Haskell; but if not you will need to execute:

```
      cabal install HUnit
```

on your terminal.
To use HUnit in a ghci session, or to specify some unit tests in a Haskell source file, you will need the following import statement:

```
      import Test.HUnit
```

## Simple Test Cases

Suppose we have a function under test `getHelloMessage :: String -> String` that greets the person provided as a String parameter value by returning "hello <person>". We can write some simple assertions as follows:

```
import Data.List
test1 = TestCase $ assertBool "hello is missing" $
    "foo" `isInfixOf` (getHelloMessage "foo")
test2 = TestCase $ assertBool "jeremy is missing" $
    "foo" `isInfixOf` (getHelloMessage "jeremy")
test3 = TestCase $ assertEqual "expect two words in greeting"
    (length $ words getHelloMessage  "world") 2
```

## Built-in Assertions

As with JUnit, we have some standard assertions we can use:

```
assertBool :: String -> Bool -> Assertion
assertString :: String -> Assertion
assertEqual :: (Eq a, Show a) => String -> a -> a -> Assertion
```

The String parameter in each case is an error message to show if the test fails. `assertBool` fails if the provided Boolean value is false. `assertString` fails if the provided String value is not null. `assertEqual` fails if the two provided values are different.

## Running our Test Cases

There is a standard test harness utility in HUnit that will run a set (actually a list) of test cases and report any failures. We need to use the data constructors `TestList` for the set of test cases and `TestLabel` for each test case function. Below is an example for the three tests we defined earlier.

```
tests = TestList [TestLabel "test1" test1,
                  TestLabel "test2" test2,
                  TestLabel "test3" test3]
```

Once we have our **`TestList`** instance, we can run this with the test controller function **`runTestTT`**:

```
runTestTT tests
```

This prints a report to standard output stating how many tests passed, with full details about individual test failures.

## A Domain-Specific Language for HUnit

To avoid repetition of keywords and other boilerplate code, HUnit has a concise domain-specific language (DSL) which we can use instead of the long-hand function names. The DSL relies on punctuation – effectively pre-defined infix operators from the Test.HUnit library. The **`assertBool`** function is replaced with the **`(@?)`** operator and **`assertEqual`** with **`(@=?)`**. For instance, we could rewrite some of our simple test cases above as follows:

```
test1 = "foo" `isInfixOf` (getHelloMessage "foo") @? "hello is missing"
test3 = 2 @=? (length $ words getHelloMessage  "world")
```

## Different styles of testing

We have now seen two testing frameworks in Haskell:

1. QuickCheck is a property-based testing framework. We supply invariants, and QuickCheck uses randomly generated (and correctly typed) inputs and reports whether the invariants are ever broken.
2. HUnit is a unit testing framework. We supply test cases – effectively invariants and their inputs, and HUnit reports whether the test cases hold for the specified inputs.

Which framework do you think is more powerful and why?

## Now it's your turn

Suppose you have written a function **`maxAndCount :: [Int] -> (Int,Int)`** which takes a list of integers and returns the maximum value in that list, and the number of times the maximum value occurs. **`maxAndCount []`** returns **`(minBound,0)`** since an empty list has no maximum value.

Devise a set of simple unit tests for **`maxAndCount`**, and code these up using HUnit.

## Further reading

There is more info about HUnit on the haskell.org website at http://hackage.haskell.org/package/HUnit and an interesting tutorial at http://jasani.org/posts/unit-testing-with-hunit-in-haskell-2007-12-05/
The *Tasty* library in Haskell provides a more elaborate testing framework.

**Functional Programming in Haskell – Lecture 10 – Wed 30 Oct 2019**

## Maybe

Sometimes, there isn't a sensible value to return for a function invocation … what is the head of an empty list? or what is the real square root of a negative number? … or how do you scrape a webpage that gives a 404 error? In such cases, the Nothing value from the Maybe datatype is highly useful

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

Effectively we are extending the set of values in type variable a, by adding one additional value that indicates the absence of a meaningful value. We can define our own functions that return Maybe values – try defining

```
safeHead :: [a] -> Maybe a


safeTail :: [a] -> Maybe [a]
```

Using Maybe values ensure functions are always productive – they always return a value for every input … they do not throw exceptions (which generally kill a Haskell program).

There are a few utility functions for dealing with Maybe values, in a specific library:

```
import Data.Maybe
```

## Introducing Functors

Given some data (like an `Int` value) wrapped up in a structure (like a `Maybe`), how do we apply a function to the wrapped `Int` value? The answer is to use `fmap`, which operates on the `Functor` typeclass.

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

Try:

- `fmap (+1) (Just 3)`
- `fmap (*2) [1,2,3]`
- `fmap (fmap Data.Char.toUpper) getLine`

Let's define our own instance of `Functor`, with a custom `fmap` routine.

```
data MagicBox a = MagicBox a deriving (Read,Show,Eq,Ord)
instance Functor MagicBox where
  fmap f (MagicBox x) = MagicBox (f x)
```

Then try

- `fmap not (MagicBox True)`

Note that the infix version of `fmap` may bewritten as `<$>` e.g. `(+1) <$> [1,2,3]`
To summarise, a Functor is a *map-able* type.

The **`Functor`** typeclass is used for types that can be mapped over. Instances of **`Functor`** are expected to follow these rules:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

**Now let's look at Applicatives**

Suppose we have a function (like **`(+1)`**) wrapped up inside a structure (like a **`Maybe`**). How do we apply this function to values inside other **`Maybe`** structures? The answer is the **`Applicative`** typeclass, particularly the **`<*>`** "apply" function.

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

try:

- **`(Just (+1)) <*> (Just 2)`**
- **`[(*3)] <*> [1..5]`**

The **`Applicative`** typeclass requires another function, called **`pure`**, which embeds a value inside a structure.

```
pure :: Applicative f => a -> f a
```

try:

- **`pure 3 :: [Int]`**
- **`pure "foo":: Maybe [Char]`**

In fact, the **`fmap`** function of the **`Functor`** may be defined in terms of **`pure`** and **`<*>`** ---can you see how?

It's possible to 'lift' two-operand functions into **`Applicative`** structures, e.g. using the utility **`liftA2`** function.
try:

- **`import Control.Applicative`**
- **`liftA2 (+) (Just 1) (Just 2)`**
- **`liftA2 (+) [1,2,3] [4,5,6]`**

**`liftA2`** can be defined in terms of **`pure`** and **`<*>`** ---can you see how?

There are some standard laws that all **`Applicative`** instances must respect:

*identity*
```
      pure id <*> v = v
```
*composition*
```
      pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```
*homomorphism*
```
      pure f <*> pure x = pure (f x)
```
*interchange*
```
      u <*> pure y = pure ($ y) <*> u
```

**Functional Programming in Haskell – Lecture 11 – Mon 4 Nov 2019**

## Monads

`Monad` is a type class in Haskell, that provides a systematic way of sequencing operations. While there are formal, mathematical frameworks for monads, we are going to start by considering monads as a way to encapsulate imperative style programming and real-world interaction. Eventually, achieving a genuine understanding of monads will feel like reaching functional programming nirvana.

## Monadic Bind

All Monads are Applicatives, and all Applicatives are Functors. This typeclass inheritance is now baked into the Haskell standard library. The key extra function in a Monad instance is the "bind" function, written `>>=`. The behaviour of `>>=` is related to `fmap` and `<*>`. Let's compare their types:

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

So the bind function takes a value embedded inside a structure, and a function that expects the non-embedded value, then returns a new embedded value. Let's demonstrate this with a simple example on Lists.

```
   [1,2,3] >>= (\n -> [(show n), "mississippi"])
```

Do you see that the lambda takes an Int input, and returns a List of Strings? So when we bind this with an Int List, we get back a (longer) List of Strings.

## The IO Monad

One way of thinking about monads is that they allow scoping of effects within the type system. The IO Monad allows us to 'contain' side-effects, or identify non-pure code from the type signature of that code.

Here are some useful functions that operate 'in the IO monad'.

```
      getLine :: IO String  -- not actually a function!
      putStrLn :: String -> IO( )
      main :: IO ()  -- not actually a function!
      readFile :: FilePath -> IO String
      writeFile :: FilePath -> String -> IO ()
      randomIO :: Random a => IO a
      getCurrentTime :: IO UTCTime -- not actually a function!
```

Let's think about some example code fragments that use these functions.

## Monadic do notation

Very quickly, we discover the need to write 'imperative' style code that will sequence IO-based computation for us. The `do` block syntactic sugar provides a convenient way to do this. (In the next lecture we will undo the syntactic sugar …)

```haskell
import Data.Time.Clock
import Data.Time.Calendar


currentYear = do
  x <- getCurrentTime
  let d = utctDay x
  let (y,_,_) = toGregorian d
  return y
```

There are a few important features to discuss here:

1. What is the difference between `x <-` … and `let x =` … inside the `do` block?
2. What is the type of the overall `do` block?
3. What does the `return` function achieve?

**Functional Programming in Haskell – Lecture 12 – Wed 6 Nov 2019**

## Web Scrapin

*Scalpel* is a web scraping framework for Haskell, that provides a systematic way of scraping text from HTML web pages. It's equivalent in functionality to the Python *Scrapy* library. While the details of Scalpel are not examinable, they are essential for the assessed coursework exercise. You will be creating a scraper tool that uses the Scalpel library.

## Install Scalpel

Scalpel is a declarative, monadic library for web scraping. It works in a similar way to the Parsec library we used in week 4 of our online course. First, we need to install the Scalpel package on our local Haskell system.

```
$ mkdir scrapingTutorial
$ cd scrapingTutorial
$ cabal sandbox init
$ cabal update
$ cabal install scalpel
```

The `scrapingTutorial` directory will be our workspace for this project. We are using it as a cabal sandbox, which means any packages we install are only visible in this directory. You will notice that the installation of scalpel requires several dependent packages to be installed.

## First Scraper

To use the Scalpel API, we need to import the module. Create a source file called simpleScraper.hs and add these lines at the start:

```
{-# LANGUAGE OverloadedStrings #-}
import Text.HTML.Scalpel
```

Now define a String value with some simple HTML markup:

```
   exampleHtml :: String
   exampleHTML = "<html><body><ul><li>Hello</li><li>World</li></ul></body></html>"
```

Now define a simple scraping function as follows. This function will return a list of Strings that are enclosed by <li> tags.

```
scrapeAllItems :: String-> Maybe [String]
scrapeAllItems input = do
  scrapeStringLike input items
    where
      items :: Scraper String [String]
      items = texts "li"
```

Finally define a main function that calls scapeAllItems:

```
main = do
  let list = scrapeAllItems exampleHTML
  print list
```

Now let's run this code. From the bash prompt in the current directory, type cabal repl to access GHCi in your sandbox environment. Then load and run your scraper.

Some things to check:

- What happens if there are no list items in the HTML?
- What happens if we have nested lists?
- What if we only want the first instance of a list item?

**Online scraping**

Now let's scrape text from a real webpage, rather than a String value. The xkcd comic has a funny subtitle, which is revealed when you mouseover the image. Let's extract the subtitle for the latest comic from the HTML source.

To be precise, the subtitle is the string contained in the `title` attribute of the `img` tag in the `div` with `id` labelled `comic`. We can use Scalpel functions to select this string from the webpage directly. Create a file called `comicScraper.hs` inside your `scrapingTutorial` directory. Insert the following Haskell source code:

```
{-- # LANGUAGE OverloadedStrings #--}
import Text.HTML.Scalpel

main :: IO ()
main = do
  res <- scrapeURL "http://xkcd.com" scrapeComic
  print res

scrapeComic :: Scraper String String
scrapeComic =
  chroot ("div" @: ["id" @= "comic"]) scrapeTitle

scrapeTitle :: Scraper String String
scrapeTitle = (attr "title" "img")
```

Note that we have two nested scraping functions here. The **`scrapeComic`** function finds the first DIV block that has the specific ID. It's possible to use other selector functions too — check out the documentation for Scalpel. The **`scrapeTitle`** function fetches the string in the TITLE attribute of IMG tags in the relevant DIV block. Note that, once we have selected a block, it is possible to get access to the raw String data. Use the **`text`** function to get the text enclosed in a block (as a String) and the **`html`** function to get the HTML source enclosed in a block. So for instance, we could modify the above **`scrapeTitle`** function to return the full HTML of the matching image:

```
scrapeTitle :: Scraper String String
scrapeTitle = do
  str <- html "img"
  return str
```

**Functional Programming in Haskell – Lecture 13 – Mon 11 Nov 2019**

## Modules

The most basic utility types, typeclasses and functions are defined in the Haskell *Prelude* module, which is available by default at top-level for all Haskell source code – it's a little like `libc` or the `java.lang` package.

For more exotic library code, we need to import other Haskell modules. We can explore published modules via the hackage site or the `cabal list` command. We need to be aware that the module name on hackage (which we use to install the module) is different to the fully qualified name in the source code (which we use to access functionality from the module). For instance, the emoji package is installed with cabal install emoji. It is 'addressed' in source code as `Data.Emoji`

```
import Data.Emoji
import Data.Maybe
main = do
    putStrLn $ fromJust $ unicodeByName "pizza"
```

## Import Statement Variants

The import statement is used to open module namespaces in the source code. There are various ways to do this:

1. `import Module` --- imports every element of the module to the top-level namespace
2. `import qualified Module` --- imports every element of the module, but they need to be accessed as `Module.element`
3. `import qualified Module as M` --- imports every element of the module, but they need to be accessed as `M.element` (M is the source code alias for Module)
4. `import Module (foo,bar)` --- only imports the foo and bar elements of the module to the top-level namespace
5. `import Module hiding (foo,bar)` --- imports every element of the module to the top-level namespace except foo and bar

Other variations are possible – e.g. qualified imports with hiding. These different import statements are useful to avoid aliasing (e.g. `Data.Map.map` and `Prelude.map`).

## Introducing Stack

Stack is a user-friendly build tool for Haskell. It does module dependency management (like pip or apt-get) and dependence-based compilation (like make or ant). We are using Stack for our coursework exercise.

First check stack is installed (`which stack`) then its version (`stack --version`) then upgrade to the latest version (`stack upgrade`). Finally update the package list (`stack update`). Now you should be ready to go!

Note that Stack (like nix) aims for reproducible builds – this means that you should be able to generate the same code regardless of when and where you run Stack on a project. This requires careful metadata handling.

One more thing – Stack performs isolated changes – so configuration is per-project and does not affect the whole system. For instance:

- you can use a different version of the GHC toolchain with each project
- you can install different modules, and different module versions for each project

## My First Stack Project

In a working directory, let's set up a new stack project called hello using the default project template.

```
stack new hello new-template
```

This creates a stack project in the hello subdirectory. You will notice a number of files in here – some Haskell files (including Main.hs in app, Lib.hs in src and Spec.hs in test) and some yaml config files. Now let's do an initial build of the project:

```
stack build
```

This takes some time since it will download appropriate binaries – for GHC, also libraries. Warning – like python pip – you need lots of filestore and quota to run stack. It puts massive binary files in your home space – normally in a hidden directory called `~/.stack`.

Once the build is complete, we can run the program (`stack run`) or the test suite (`stack test`). When we make source code modifications, stack automatically runs dependence resolution and recompiles code as necessary. Compiled binaries for the project are stored in the hidden directory `.stack-work` inside the project directory.

## Extra Dependencies

Suppose we want to use the emoji libraries we explored earlier. We need to register this with stack, by adding a line in the package.yaml file:

```
dependencies:
- base >= 4.7 && < 5
- emoji == 0.1.0.2
```

Because the emoji library isn't in the curated set of packages for Stack, we also need another line in stack.yaml:

```
extra-deps:
- emoji-0.1.0.2
```

**Functional Programming in Haskell – Lecture 14 – Wed 13 Nov 2019**

## More on Monads

We have already looked at the `Monad` typeclass in Haskell; it's a structure for sequencing actions (particularly the `IO` Monad). Today we are going to delve into more detail about Monads.

## Monad Functions

According to the `Monad` typeclass specification, a `Monad` type `m` must provide implementations of three characteristic functions:

```
return :: a -> m a                      -- embed
(>>)   :: m a -> m b -> m b             -- sequence
(>>=)  :: m a -> (a -> m b) -> m b      -- bind
```

In fact, the default implementation of `>>` is defined in terms of `>>=` so it's not necessary to provide the `>>` definition when typeclassing `Monad`.

## Syntactic Sugar

The core Haskell language is actually very simple. Many of the higher-level constructs we have seen (for instance, `where` clauses) can be rewritten (de-sugared) into more simple syntax. The monadic `do` block is actually just syntactic sugar.

## Monad Laws

Like Functors and Applicatives, there are laws that govern the behaviour of functions in the Monad typeclass. All implementations of Monad should respect these laws.

*Right Identity*

```
m >>= return        ==    m
```

*Left Identity*

```
return x >>= f      ==    f x
```

*Associativity*

```
(m >>= f) >>= g     ==    m >>= (\x -> f x >>= g)
```

## Utility Functions

The base library `Control.Monad` defines a number of useful functions for monadic computatation. These include:

**`liftM`**
**`liftM2`**
**`liftM`***<n>*

**`mapM`**
**`mapM_`**

**`replicateM`**
**`replicateM_`**

and plenty more … check them out on Hackage.