NOTES ON HASKELL

VAIBHAV KARVE

These notes were last updated September 1, 2018. They are notes taken from my reading of:

- Learn You a Haskell for Great Good! by *Miran Lipovača*.
- Reading Simple Haskell
- Haskell Programming from First Principles (2017) by Chris Allen, Julie Moronuki.

Contents

1. Introduction

(1) There are three ways to run Haskell programs:
- We can run commands interactively by typing `ghci` into terminal. This starts an interactive GHC session (GHC is a Haskell compiler). To change the prompt text, we type in –
  `:set prompt "ghci> "`
- We can write the program into `.hs` files and then compile them usign GHC at a terminal. This creates a `.out` output file that can then be executed. The contents of a `myfunctions.hs` program can also be loaded into the `ghci` session by typing
  `:l myfunctions`
  provided this `.hs` is in the same folder as the one in which `ghci` was invoked. For reloading a `.hs` file, we can use either `:l myfunctions` or simply `:r` as reload commands.
- We can use Jupyter notebook's IHaskell kernel in order to run Haskell commands interactively in a notebook.

In what follows, we write commands assuming we are using a Jupyter notebook and each line in a cell-block is prefaced with a `>` symbol to differentiate input from output.

(2) Haskell is a *purely functional language*.

(3) A variable cannot be set to one value and then have its value changed.

```
> a = 5
> a = "some string"
[ErrorMessage]
```

(4) The philosophy is that we don't tell the computer what to do (as as in imperative languages), but instead tell it what a thing *is*.

(5) Variables and functions have no side-effects i.e. the only thing a function can do is calculate something and return it as a result.

(6) If a function is called twice with the same parameters, then it is guaranteed to return the same results. This is called *referential transparency*.

(7) Haskell has *lazy implementation* i.e. it doesn't calculate/evaluate anything unless specifically told otherwise. This also allows it to have infinite data structures.

(8) Lazily transforming composition of functions is really efficient. Suppose `f` is a function that acts on `my_list` and returns another list. Then, calling `f(f(f(my_list)))` in a non-lazy results in the calculation of `f(my_list)`, `f(f(my_list))` and then `f(f(f(my_list)))` in order. This results in 3 passes through `my_list` to compute the result.

However, a lazy implementation calculates the i^{th} entry of the result as `f(f(f(my_list[i])))` and thus makes only one pass through `my_list`.

(9) Haskell is *statically typed* i.e. during compile time, the compiler knows the type of each variable. The compiler knows this either because the programmer specifies the type of certain variables or it may infer the type of other variables through Haskell's system of *type inference.*

(10) Historically, development of Haskell began in 1987. In 2003 the Haskell Report was published, which defines a stable version of the language.

2. Basics

(1) All arithmetic operations of `+, -` and `*` work as usual.

(2) Division results in a floating point answer. Integer division can be done by `div`.

```
> 5 / 2
2.5

> div 10 4
2
```

(3) Put parenthesis around negative numbers.

```
> 5 * -3
[ErrorMessage]

> 5 * (-3)
-15
```

(4) `&&`, `||` and `not` represent AND, OR or and NOT operators.

(5) Only compare variables of the same type.

```
> 5 == 4
False
```

```
> 5 /= 4
True

> 5 == "4"
[ErrorMessage]

> 5 /= "4"
[ErrorMessage]
> 5 == 5.0
True

> 5 == 5.
[ErrorMessage]
```

(6) *Infix functions* like `+` and `*` take arguments on both sides. Functions that are not infix are *prefix functions*. Examples:

```
> succ 8
9

> min 8 10
8
```

(7) Function arguments are separated by space. Space takes precedence over all other operations.

```
> succ 9 * 10
100

> succ(9 * 10)
91

> succ 9 + max 5 4 + 1 == (succ 9)+ (max 5 4)+ 1
True
```

(8) Prefix functions can be turned into infix functions by using backticks to write `div` :

```
> div 10 4
2

> 10 `div` 4
2
```

(9) In order to define a function, we just state its action on `x`:

```
> f x = x + x
> f 12
24

> g x y = 2*x + 2*y
> g 3 4
14
```

(10) Function call is left-associative.

(11) One can also define operators:

```
> x +- y = (x + x)- (y + y)
```

(12) If statements can be written as follows:

```
> f x = if x > 100
```

```
        then x
        else 2*x
```

Note that the else statement is mandatory because every function must return something.

```
> f 4
8

> f 400
400
```

(13) One can operate on the output value of an if condition before returning it.

```
> f' x = (if x > 100 then x else 2*x)+ 1
```

Apostrophes are allowed characters in function names. However, function names can never begin with upper-case letters.

(14) A function which takes no arguments is a *definition* or a *name*.

```
> f = "Just a string"
> f
"Just a string"
```

(15) We can name part of the computation using `let..in` or `where`.

```
sumOf3 x y z =
  let temp = x + y
  in temp + z

sumOf3' x y z = temp + z
  where temp = x + y
```

3. Lists and List Comprehension

(1) Lists are a *homogenous* data structure and hence all the entries in a list must have the same type.

(2) Characters are encased in single quotes (''). Strings are encased in double quotes. Strings are just lists of characters (the double quotes are just syntactic sugar).

```
> 'a'
'a'

> 'ab'
[ErrorMessage]

> "abc" == ['a', 'b', 'c']
True

> "a" == 'a'
[ErrorMessage]

> "a" == ['a']
True
```

(3) Since strings are lists, all the list functions can be used on them.

(4) List (and string) concatenation is done by ++

```
> [1,2,3] ++ [4, 5]
[1,2,3,4,5]
```

(5) The `++` operator runs through all the elements in the list on the left side. Hence it should be avoided with long lists as it is inefficient. Prepending with a given element is faster and more efficient (nearly instantaneous). Prepending can be done with the *cons operator*, written as `:`.

```
> 'a' : "cat"
"a cat"

> 5:[1,2,3,4]
[5,1,2,3,4]
```

(6) `[]` is an empty list. `[1,2,3]` is actually syntactic sugar for `1:2:3:[]`. Obviously, `[]`, `[[]]` and `[[],[],[]]` are all different (but valid) things.

(7) List indexing starts at `0`. List entries can be called by using the `!!` operator.

```
> "Mumbai"!! 3
'b'
```

(8) Asking for the 0^{th} element of a list raises a suggestion that `head8 9.9626 Tf 11.493..2425 0 Td [(should)-333`

`)]TJ001rg001RG[Mb'`

```
> reverse "Mumbai"
"iabmuM"
```

(14) In order to check if a list is empty, use `null` instead of `mylist == []`.

```
> null [1,2,3]
False

> null []
True
```

(15) `take` extracts a specified number of elements from the beginning of the list (wherever possible).

```
> take 3 "Mumbai"
"Mum"

> take 100 "Mumbai"
"Mumbai"

> take 0 "Mumbai"--and Haskell suggests we simply use [] instead
[]

> take -1 "Mumbai"
[ErrorMessage]
```

(16) `drop` is the same, except it drops the given elements from the beginning of the list and returns the remaining list.

(17) `maximum` and `minimum` return the largest and smallest values (while keeping in mind the lexicographic ordering).

(18) `sum` returns the sum for a list of numbers; `product` the product.

(19) `elem` tells us if a particular element is in a list (provided the types are appropriate).

```
> elem 'a' "Mumbai"
True

> elem 0 in "Mumbai"
[ErrorMessage]
```

(20) *Ranges* are a way to make lists that are arithmetic sequences of elements that can be enumerated. Both numbers and characters can be enumerated.

```
> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

(21) Step sizes can be specified for ranges.

```
> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]

> [20..1] --does NOT create a reverse list
[]

> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

```
> [13,2*13..24*13] --this is one way to obtain the first 5 multiples of 13.
[13,26,39,52,65]

> take 5 [13,2*13..] --this is another way using infinite ranges
[13,26,39,52,65]
```

(22) It is advised to not use floats inside ranges because it has issues with rounding errors.

(23) Simply creating infinite lists like `[1..]` seems to be a bad idea and falls into an infinite loop, thus killing the IHaskell kernel.

(24) `cycle` takes a list and cycles it into an infinite list. `repeat` takes an element amd produces an infinite list of just that element.

```
> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]

> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]

> replicate 3 10 --a better way of replicating things.
[10,10,10]
```

(25) List comprehension in Haskell:

```
> [2*x | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]

> [2*x | x <- [1..10], 2*x >= 12]
[12,14,16,18,20]
```

(26) This process of weeding out list elements based on some predicate (condition) is also called *filtering*.

(27) The *fizzbuzz* example:

```
f x
  | x `mod` 15 == 0 = "fizzbuzz" -- we use guards "|" to avoid nested if-else.
  | x `mod`  3 == 0 = "fizz"
  | x `mod`  5 == 0 = "buzz"
  | otherwise = show x           -- show allows us to cast each number as a string
      .

> [f x | x <- [1..30]]
["1","2","fizz","4","buzz","fizz","7","8","fizz","buzz","11","fizz","13","14","fizzbuzz","16","17",
"fizz","19","buzz","fizz","22","23","fizz","buzz","26","fizz","28","29","fizzbuzz"]
```

(28) In list comprehension, we can use predicates, along with if-then-else statements.

```
> g xs = [if x < 10 then "BOOM!"else "BANG!"| x <- xs, odd x]
> g [8..16]
["BOOM!","BANG!","BANG!","BANG!"]

> [2*x | x <- [1..10] , odd x, x /= 5] --multiple predicates, all separated by commas
[2,6,14,18]

> removeLowerCase string = [c | c <- string, c `elem` ['A'..'Z']] --removes lowercase characters

> [x^y | x <- [4,5,6], y <- [0,1,2,3]] --and one can draw from multiple lists.
[1,4,16,64,1,5,25,125,1,6,36,216]
```

```
> names = ["Jon", "Dany", "Tyrion", "Aegon"]
> houses = ["Stark", "Targ", "Dayne", "Blackfyre"]
> [name ++ "" ++ house | name <- names, house <-   names
```

```
(13,'c'),(14,'h'),(15,' '),(16,'c'),(17,'h'),(18,'a'),(19,'r'),(20,'a'),(21,'c'),(22,'t'),
(23,'e'),(24,'r')]
```

This works because `zip` truncates when it exhausts the smaller list.

5. Types

(1) Haskell has a static type system, meaning that the type of every expression is known at compile time. However, Haskell is also globally type inferred, meaning Haskell can figure out type signatures on its own and in every case, Haskell will choose the most general type signature for us.

(2) A *type* is a kind of label every expression has. We can find out the type by using the `:t` command.

```
> :t 'a'
'a' :: Char

> :t "a"
"a" :: [Char]

> :t (True, 'a')
(True, 'a'):: (Bool, Char)

> :t 4 == 5
4 == 5 :: Bool
```

(3) `::` is to be read as "has type of".

(4) Explicit types always begin with an uppercase letter.

(5) Functions also have types. When writing functions, it is generally considered a good practice to specify type declarations for that function (unless it is a very short function). For example,

```
f :: Int -> Int
f x = (if x >100 then x else 2*x) + 1

-- [[parameters are separated by -> with no special distinction between parameters
     and return types]]
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

(6) The operator `->` is right-associative.

(7) `Int` type is for bounded integers. On a 64-bit system, `Int` values can typically range from -9223372036854775808 to 9223372036854775807 (that's about 19 digits).

(8) `Integer` type is also for integers, but is not bounded and can therefore be used for really big numbers. `Int` however is more efficient that `Integer`.

```
factorial :: Int -> Int
factorial n = product [1.. n]

factorial' :: Integer -> Integer
factorial' n = product [1.. n]
```

```
> factorial 50 --gives the wrong output
-3258495067890909184
> factorial' 50
30414093201713378043612608166064768844377641568960512000000000000
```

(9) `Float` is a real floating point with single precision. `Double` is double the floating point with double the precision.

(10) `pi` is a predefined quantity in Haskell for some reason.

```
> pi
3.141592653589793
```

(11) `Bool` is the boolean type and can have only two values – `True` and `False`.

(12) `Char` represents a character and is denote by single quotes.

(13) There are an infinite number of tuple types because the type fo a tuple is dependent on length. `()` is the empty tuple of type `()`. The type `()` has only this one value – `()`.

(14) Functions that have type variables in them are *polymorphic functions*.

```
> :t head --head returns the first element in a list
head :: [a] -> a

> :t fst --fst returns the first component in a pair
fst :: (a,b)-> a
```

(15) `type` can be used to give a new alias to an existing type. The two names can then be used interchangeably.

```
type Number = Int
x :: Number
x = 42
```

(16) New types can be defined using the keyword `data`.

```
> data Variable = Variable --This lets us define a type called Variable without saying what Variable is
```

(17) Sum types are the "disjoint union" of smaller types. In other words, for defining sum types we simply enumerate all the possible types that it can be.

```
data Information
  = String
  | Int

whoami :: Information
whoami = "Batman"

myage :: Information
myage = 104
```

(18) We could define a crazier sum type:

```
data Occupation
  = Mathematician
  | Programmer
  | Painter
```

Note: `Mathematician`, `Programmer` and `Painter` are not predefined types. So, we defined a new type (`Occupation`) at the same time that we defined a bunch of smaller types. Recall that type names must always start with an uppercase letter.

(19) Product types are types made by compounding data of existing types.

```
data Pairings
  = MakePairs String Int
```

```
x :: Pairings
x = MakePairs "Batman" 104
```

Here, `MakePairs` is what is called a value constructor or "tag". We could replace it with any other name – say like `Tag` or `MakePairings` or `Xyz` – it doesn't matter as long as we use some value constructor.

(20) It is possible to mix Sum and Product types.

```
data Color =
  Red
  | Blue
  | Green
  | RGB Int Int Int

blue :: Color
blue = Blue -- this works because Blue is a type having only one value, also named
     Blue

magenta :: Color
magenta = RGB 255 0 255
```

In fact, in this example, there are 4 value constructors – `Red`, `Blue`, `Green` and `RGB`. The first three don't have other types following them so they make a singleton value.

(21) Using records, one can name the fields in a product type:

```
data RGB = MkRGB
  { rgbRed   :: Int
  , rgbGreen :: Int
  , rgbBlue  :: Int
  }

red :: RGB
red = MkRGB
  { rgbRed   = 255
  , rgbBlue  = 0
  , rgbGreen = 0
  }
```

(22) Polymorphic identity function can be defined by using a type variable `a`:

```
identity :: a -> a -- works for any type
identity x = x

seven :: Int
seven = identity 7
```

(23) We can also define funtion composition:

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

f . g = compose f g
```

(24) A recursive datatype is a data definition that refers to itself. This lets us define data structures like linked lists and trees.

```
data IntList
  = EndOfIntList
  | ValAndNext Int IntList

-- the list [4, 0, 5]
```

```
list405 :: IntList
list405 = ValAndNext 4 (ValAndNext 0 (ValAndNext 5 EndOfIntList))
```

(25) A typeclass is a sort of interface that defines some behavior. If a type is part of a typeclass, that means that the type supports and implements the behavior the typeclass defines.

```
> :t (==) -- what is the type signature of the equality operator?
(==) :: (Eq a)=> a -> a -> Bool
```

Note that if a function is comprised of only special characters, it is considered an infix function by default. Hence to examine its type we need to surround it with parenthesis.

(26) Also note that this was not the real output of `:t (==)`. The output mentioned above is what we get from GHCI. The output mentioned below is slightly more complicated and is what we get from the Haskell kernel on Jupyter:

```
> :t (==)
(==) :: forall a. Eq a =>a -> a -> Bool
```

(27) Everything before the `=>` symbol is called a *class constraint*.

(28) The type declaration of `==` can be read as: the equality function takes any two values that are of the same type and returns a `Bool`. The type of those values must be a member of the `Eq` class.

(29) The `Eq` typeclass provides an interface for the testing of equality. All standard Haskell types except for functions and except for IO (the type for dealing with input output) are in the `Eq` typeclass.

(30)

```
> :t elem --elem return True if argument1 is an element of argument2
elem :: (Eq a, Foldable t)=>a -> t a -> Bool
```

(31) `Eq` is a typeclass used for types that support equality testing. All members of `Eq` implement `==` and `/=`.

(32) `Ord` is a typeclass for types that have ordering.

```
> :t (>)
(>) :: Ord a =>a -> a -> Bool
```

All standard types except functions are members of `Ord`. `Ord` covers all the standard comparing functions such as `>`, `<`, `>=` and `<=`.

(33) The `compare` function takes two `Ord` members of the same type and returns an ordering, where `Ordering` is a type that can be `GT`, `LT` or `EQ`, meaning greater than, less than and equal to respectively. Note that this `EQ` (which is an element of type `Ordering`) is different from `Eq` (which is a typeclass).

```
> :t compare
compare :: Ord a =>a -> a -> Ordering


> compare 4 5
LT
```

(34)

(35) To be continued...