NOTES ON HASKELL

VAIBHAV KARVE

These notes were last updated July 24, 2018. They are notes taken from my reading of Learn You a Haskell for Great Good! by *Miran Lipovača*.

1. Starting out

(0) There are two ways to run Haskell commands:
- We can run it by typing `ghci` into terminal. This starts an interactive IHaskell session. To change the prompt text, we type in –
  `:set prompt "ghci> "`
- We can use Jupyter notebook's IHaskell kernel in order to run Haskell commands in a notebook.

In what follows, we write commands assuming we are using the latter method. All lines in cell-blocks are prefaced with a `>` symbol to differentiate input from output.

(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) 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
```

(11) 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.

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

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

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

(14) 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
```

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

(16) List (and string) concatenation is done by `++`

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

(17) 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]
```

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

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

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

(20) Asking for the 0^{th} element of a list raises a suggestion that `head` should be used instead.

```
> head "Mumbai"
'M'
```

(21) Lists of lists are allowed. Lists within lists can be of different lengths but not of different types. Therefore, lists within lists is allowed as long as they all have the same level of nesting.

(22) Lists can be compared if the stuff they contain can be compared (this includes strings). Comparison operators are `<`, `>`, `<=`, `>=`, `==` and `/=`. Comparison of lists is done in lexicographic order!

```
> [3,2,1] > [2,1,0]
True

> [3,2,1] > [2,10,100]
True

> [3,4,2] > [3,4]
True
```

```
> "abc"> "acd"
False
```

(23) `tail` return everything but the head of the list. `last` returns the last element. `init` returns everything but the last element.

```
> tail "Mumbai"
"umbai"

> last "Mumbai"
'i'

> init "Mumbai"
"Mumba"
```

(24) Asking for head, tail, init or last of empty list `[]` raises an exception. This error cannot be caught at compile-time and so one should watch out for it.

(25) `length` returns length. `reverse` reverses.

```
> length "Mumbai"
6

> reverse "Mumbai"
"iabmuM"
```

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

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

> null []
True
```

(27) `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]
```

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

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

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

(31) `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]
```

(32) To be continued...