Haskell Basics

Piotr Karamon

Contents

1 Comments

```
-- Single line comment
{-
Multiline
comment
-}
```

2 Basic types

2.1 Int

`Int` can store whole numbers in range $[-2^{63}, 2^{63})$

```
minInt = minBound :: Int
maxInt = maxBound :: Int
"Lower bound " ++ show minInt ++ " Upper Bound " ++ show maxInt
```

```
Lower bound -9223372036854775808 Upper Bound
↪   9223372036854775807
```

2.2 Integer

`Integer` is an unbounded whole number. Works like the `int` type in python.

```
x = 2 ^ 124 :: Integer
"Very big number " ++ show x
```

```
Very big number 21267647932558653966460912964485513216
```

2.3 Floating point numbers

`Float` - single precision floating point numbers. `Double` - double precision floating point numbers. In reality you pretty much always should use `Double`

```
x = (3.14 * 2.0 + 2.71) :: Double
x
```

```
8.99
```

2.4 Char

Single quotes ''

```haskell
firstInitial = 'P'
secondInitial = 'K'

show firstInitial ++ " " ++ show secondInitial
```

```
'P' 'K'
```

There are also typical math functions `sin`, `cos`, `tan`, `asin`, `atan`, `acos`, `sinh`, `tanh`, `cosh`, `asinh`, `atanh`, `acosh`.

```haskell
9 ** 2
9 ** (0.5)
exp 1
log (exp 1)
log 1024 / log 2
```

```
81.0
3.0
2.718281828459045
1.0
10.0
```

```haskell
truncate (-3.5) -- discards the fractional part
floor (-3.5) -- finds the biggest integer smaller than -3.5
"---"
round 9.70
round 9.5
round 9.123
"---"
ceiling 9.00001
```

```
-3
-4
---
10
10
9
---
10
```

3 Lists

3.1 Creating lists

- lists in Haskell are unidirectional
- **we can only add items to the front of a list**

```haskell
primes = [2, 3, 5, 7]
morePrimes = primes ++ [11, 13, 17]
morePrimes

indexes = 1 : 2 : 3 : 4 : 5 : []
indexes
```

```
[2,3,5,7,11,13,17]
[1,2,3,4,5]
```

```haskell
[2,4..20]
[10,4..(-20)]
['a'..'z']
['a'..'z'] ++ ['A'..'Z'] ++ ['0'..'9']
```

```
[2,4,6,8,10,12,14,16,18,20]
[10,4,-2,-8,-14,-20]
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

```haskell
take 10 (repeat 2)
replicate 10 3
take 10 (cycle [9,2,0])
```

```
[2,2,2,2,2,2,2,2,2,2]
[3,3,3,3,3,3,3,3,3,3]
[9,2,0,9,2,0,9,2,0,9]
```

We can create nested lists.

```haskell
grid :: [[Int]]
grid = [[1,0,1], [1,1,1], [1,2,1]]
```

3.2 Useful list functions

```haskell
nums = [8,9,11,13,2]
nums !! 1
length nums
reverse nums
```

```
9
5
[2,13,11,9,8]
```

```haskell
null []
null nums
```

```
True
False
```

```haskell
last nums
init nums
take 3 nums
drop 3 nums
```

```
2
[8,9,11,13]
[8,9,11]
[13,2]
```

```
9 `elem` nums
maximum nums
minimum nums
sum nums
product nums
```

```
True
13
2
43
20592
```

```
import Data.List
sort nums
```

```
[2,8,9,11,13]
```

```
zipWith (+) [1,2,3] [4,5,2,1]
zipWith (\x y -> if(x > y) then x else y) [1,2,3] [4,5,2,1]
zipWith max [1,2,3] [4,5,2,1]
zipWith min [1,2,3] [4,5,2,1]
zipWith (\ x y -> x/y + x*x + y*y) [1,2,3] [4,5,2, 1]
```

```
[5,7,5]
[4,5,3]
[4,5,3]
[1,2,2]
[17.25,29.4,14.5]
```

3.3 List comprehensions

List comprehensions are very similar to those in Python. The blueprint is `[<EXPR> | x <- <list>, <COND 1>, <COND 2>, ...]`

```
[x * 2 | x <- [1..10]]
[x * y | x <- [1..3], y <-[1..3]]
[x*3 | x <- [1..100], x * 3 <= 50]

[x | x <- [1..500], x `mod` 2 == 0, x `mod` 17 == 0]
```

```
[2,4,6,8,10,12,14,16,18,20]
[1,2,3,2,4,6,3,6,9]
[3,6,9,12,15,18,21,24,27,30,33,36,39,42,45,48]
[34,68,102,136,170,204,238,272,306,340,374,408,442,476]
```

3.4 Map Filter fold

`map` and `filter` functions are often interchangeable with list comprehensions.

```haskell
nums = [1,2,3,4,5]
square n = n * n
map square nums
map (\x -> x*x) nums
```

```
[1,4,9,16,25]
[1,4,9,16,25]
```

```haskell
nums = [1..100]
filter (\x -> x `mod` 3 == 0 && x`mod` 10 == 3) nums
```

```
[3,33,63,93]
```

`fold` functions are similar to `reduce` functions in python/javascript.

```haskell
foldl (+) 0 [1,2,3,4] -- sum
foldl max 0 [1,2,3,4] -- maximum, stupid but works

foldl (\acc x -> acc ++ " " ++ show x ) "" [1,2,3,4]
foldl (\acc x -> show x ++ " " ++ acc ) "" [1,2,3,4]
foldr (\x acc -> acc ++ " " ++ show x ) "" [1,2,3,4]
```

```
10
4
 1 2 3 4
4 3 2 1
 4 3 2 1
```

3.5 TakeWhile

```
takeWhile (<= 20) [1..]
takeWhile (\x -> x <= 20) [1..20]
```

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

4 Strings

Strings are just lists of characters.

```
removeUppercase :: String -> String
removeUppercase st = [c | c <- st, not (c `elem` ['A'..'Z'] )]

main = do
  putStrLn $ removeUppercase "Hello THERE!"
```

```
ello !
```

5 Tuples

Similar to tuples in python.

```
john = ("John Doe", 42)
username = fst john
age = snd john
username ++ " is " ++ show age ++ " years old."
```

```
John Doe is 42 years old.
```

Usage in functions

```
isAdult (name, age) = age >= 18
isAdult john
-- we didnt specify the type of isAdult function meaning
isAdult ([3.14, 2.71], -2.1231)
```

```
True
False
```

You can create a list of tuples by using `zip` function.

```
ids = [1,2,3,4]
names = ["Bob", "Joe", "Tom", "Rob"]
zip ids names
```

```
[(1,"Bob"),(2,"Joe"),(3,"Tom"),(4,"Rob")]
```

6 Main function

Main function is the entry point of a program written in haskell. `do` keyword lets you chain functions together.

```
import System.IO

main = do
  putStrLn "What's your name? "
  name <- getLine
  putStrLn ("Hello " ++ name)
```

7 Function

7.1 Basic

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

main = do
  print (fact 5)
```

```
120
```

7.2 Guards

Guards are just fancy and convenient if/case statements.

```haskell
message age | age == 18 = "You are an adult"
            | age == 6  = "You should go to school"
            | age == 19 = "You might want to consider higher education"
            | otherwise = "Nothing fancy"
main = do
  putStrLn (message  18)
  putStrLn (message 19)
  putStrLn (message 6)
  putStrLn (message 54)
```

```
You are an adult
You might want to consider higher education
You should go to school
Nothing fancy
```

7.3 Functions with lists

```haskell
myMap :: (a -> b) -> [a] -> [b]
myMap _      []       = []
myMap mapper (x : xs) = mapper x : myMap mapper xs

myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ [] = []
myFilter predicate (x : xs) | predicate x = x : myFilter predicate xs
                            | otherwise   = myFilter predicate xs

main :: IO ()
main = do
  print (myMap (* 3) [1, 2, 3, 4])
  print (myFilter (>= 2) [1, 2, 3, 4])
```

```
[3,6,9,12]
[2,3,4]
```

```haskell
describeList :: [Int] -> String
describeList []           = "Empty"
describeList (x     : []) = "THe only element is: " ++ show x
describeList (x : y : []) = "First: " ++ show x ++ " Second: " ++ show y
describeList nums = "List contains " ++ show (length nums) ++ " elements"

main :: IO ()
main = do
  putStrLn (describeList [])
```

```
  putStrLn (describeList [3])
  putStrLn (describeList [3, 4])
  putStrLn (describeList [3, 4, 5])
```

```
Empty
THe only element is: 3
First: 3 Second: 4
List contains 3 elements
```

```
maximum' :: (Ord a) => [a] -> a
maximum' []  = error "maximum of empty list"
maximum' [x] = x
maximum' (x : xs) | x > maxtail = x
                  | otherwise   = maxtail
  where maxtail = maximum' xs

elem' :: (Eq a) => a -> [a] -> Bool
elem' a []       = False
elem' a (x : xs) = a == x || elem' a xs

main = do
  print $ maximum [1, 2, 3, 4, 2, 1]
  print $ maximum "hello world"
  print $ elem' 'e' "hello world"
  print $ elem' 1 [2,3]
```

```
4
'w'
True
False
```

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f [] _ = []
zipWith' f _ [] = []
zipWith' f (x:xs) (y:ys) = (f x y) : zipWith' f xs ys

addIndex :: Int -> String -> String
addIndex x string = show x ++ ". " ++ string

main = do
  print $ zipWith' addIndex [1,2,3] ["Bob", "Tom", "Tim", "Joe"]
  print $ zipWith' (++) ["Bob", "Tim"] ["Smith", "Johnes"]
```

```
["1. Bob","2. Tom","3. Tim"]
["BobSmith","TimJohnes"]
```

7.4 Pretty pattern matching with lists

```haskell
tell :: Show a => [a] -> String
tell []     = "Empty list"
tell [x]    = "The list has one element " ++ show x
tell [x, y] = "The list has two elements " ++ show x ++ " and " ++ show y
tell (x : y : _) =
  "This list is long. The first two elements are: "
    ++ show x
    ++ " and "
    ++ show y

main = do
  putStrLn ( tell "")
  putStrLn ( tell "c")
  putStrLn ( tell [3.14, 2.71])
  putStrLn ( tell [3.14, 2.71, 4, 5, 6])
```

```
Empty list
The list has one element 'c'
The list has two elements 3.14 and 2.71
This list is long. The first two elements are: 3.14 and
↪  2.71
```

7.5 Composition

Dot operator . is doing function composition meaning f (g x) = (f . g) x

```haskell
(putStrLn . show) (1 + 2)
(putStrLn . show) $ 1 + 2
putStrLn . show $ 1 + 2
```

```
3
3
3
```

7.6 all@

```
firstChar :: String -> String
firstChar []            = "Empty List"
firstChar all@(x : xs) = "First char in " ++ all ++ " is " ++ [x]

main = do
  putStrLn (firstChar "Bob")
```

```
First char in Bob is B
```

7.7 misc

```
areStringsEqual :: String -> String -> Bool
areStringsEqual []       []       = True
areStringsEqual (x : xs) (y : ys) = x == y && areStringsEqual xs ys
areStringsEqual _        _        = False

main = do
  print (areStringsEqual "robert" "tom")
  print (areStringsEqual "robert" "robert")
```

```
False
True
```

7.8 case

`case` statement works the best with enumeration types.

```
employeeId name = case name of
  "Robert" -> 1
  "Tim" -> 42
  _ -> -1

main = do
    print (employeeId "Robert")
```

```
    print (employeeId "Tim")
    print (employeeId "Jacob")
```

```
1
42
-1
```

Now example with pattern matching.

```haskell
describeList :: [a] -> String
describeList xs = "The List is " ++ case xs of
  []  -> "Empty"
  [x] -> "singleton list"
  xs  -> "a longer list"

main = do
  putStrLn $ describeList []
  putStrLn $ describeList [2]
  putStrLn $ describeList [1, 2, 3]
```

```
The List is Empty
The List is singleton list
The List is a longer list
```

7.9 Partial application

> When you give a function an argument it returns another function that takes 1 less argument than the original one.

You can even say that every function in Haskell takes exactly 1 argument. This whole behavior is known as "currying".

```haskell
(+2) 3
twice f x = (f.f) x
twice ("Hello " ++ ) "John"

isuppercase = ( `elem` ['A'..'Z'] )
isuppercase 'A'
isuppercase 'a'

map (/2) [1..10]
map (2-) [1..10] -- works
-- does not work: map (-2) [1..10]
map (`subtract` 2) [1..10]
```

```
5
Hello Hello John
True
False
[0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
[1,0,-1,-2,-3,-4,-5,-6,-7,-8]
[1,0,-1,-2,-3,-4,-5,-6,-7,-8]
```

Flip function takes in a function $f(x, y)$ and returns a the same function but the arguments are fliped meaning $f(y, x)$.

```haskell
flip' :: (a -> b -> c) -> b -> a -> c
flip' f x y = f y x

main = do
  print $ flip' zip [1,2,3] "abc"
```

```
[('a',1),('b',2),('c',3)]
```

7.10 Application using $

The `$` sign is as everything in Haskell just a function with the following definition.

```haskell
($) :: (a -> b) -> a -> b
f $ x = f x
```

Function application with a space is left-associative so `f a b c` is the same as `(((f a) b) c)`. Function application with $ is right-associative.

```haskell
sum $ map (*2) [1..5]
sum $ filter (>= 3) $ map (+3) [1..20]
```

```
30
270
```

Apart from reducing parentheses we can also use it call functions. For example

```haskell
map ($ 3) [(+2), (*3), (/2)]
map (\f -> f 3) [(+2), (*3), (/2)]
```

```
[5.0,9.0,1.5]
[5.0,9.0,1.5]
```

8 Modules

8.1 Creating a module

A module in haskell is just a file.

```haskell
module NameOfModule (add, multiply) where
add x y = x + y
multiply x y = x * y
```

8.2 Importing a module

When we write `import ModuleName` all of the exposed functions types etc are directly put into our namespace. This can very easily create conflicts to remedy this we have a couple options.

- import only things that you need thus not polluting the namespace as much

```haskell
import Data.List (nub, sort)
```

- by excluding entities that produce conflicts and ambiguity

```haskell
import Data.List hiding (nub, sort)
```

- qualified import this puts all of the entities under its own namespace

```haskell
import qualified Data.List
Data.List.nub [1,1,2,2,2,3,2] -- removes duplicates
```

```
[1,2,3]
```

- writing `Data.List.nub` is a bit of a pain so we can use aliases

```
import qualified Data.List as L
L.nub [1,1,2,2,2,3,2] -- removes duplicates
```

```
[1,2,3]
```

9 Enumeration types

We create enumeration by using `data` keyword

```
data VehicleType = Car
                 | Pickup
                 | Suv
                 | Minivan
                 deriving Show

vehicleDesc :: VehicleType -> String
vehicleDesc vt = case vt of
  Car     -> "Quick but rather small"
  Pickup  -> "Practical but burns a lot of fuel"
  Suv     -> "Good for offroading"
  Minivan -> "Great for families"

main = do
  print (vehicleDesc Suv)
```

```
Good for offroading
```

10 Type classes

10.1 Basics

- Type classes are for example `Num,Eq,Show`.
- they are similar to interfaces in other languages

We will define Employee.

```
data Employee = Employee {
  name :: String,
  position :: String,
  idNum :: Int
} deriving (Eq, Show)
```

Now we can use `show` `print` or `=` with our employees.

```
samSmith = Employee { name = "Sam Smith", position = "Manager", idNum = 1
↪ }
pamMarx = Employee { name = "Pam Marx", position = "Sales", idNum = 1 }
isSamPam = samSmith == pamMarx

main = do
  print isSamPam
  print samSmith
```

- When we use `derving (Eq, Show)` haskell provides us with some implementation of *show print or + methods* loosely speaking. We can however provide our own.
- This whole procedure is very similar to defining `__add__`, `__eq__`, ... methods inside of classes in python.

```
data ShirtSize = S | M | L

instance Eq ShirtSize where
  S == S = True
  M == M = True
  L == L = True
  _ == _ = False

instance Show ShirtSize where
  show S = "Small"
  show M = "Medium"
  show L = "Large"

smallAvail = S `elem`  [L, M, S]

main = do
  print smallAvail
  print S
```

```
True
Small
```

10.2 Overview of typeclasses

typeclass name	description	functions
`Eq`	equality	`==` , `/=`
`Ord`	ordering	`> <= > >=`
`Show`	displaying as a string	`show print`
`Read`	changing a string into value	`read`
`Enum`	sequentially ordered types	`[first..last] succ pred`
`Bounded`	there exist upper and lower bound	`minBound maxBound`
`Num`	numerics, can be added like numbers	`+ *`

```
dist x y = sqrt (x * x + y * y)

data Point = Point
  { x :: Double
  , y :: Double
  }

instance Eq Point where
  (Point ax ay) == (Point bx by) = dist ax ay == dist bx by

instance Ord Point where
  (Point ax ay) `compare` (Point bx by) = dist ax ay `compare` dist bx by

instance Show Point where
  show (Point ax ay) = "(" ++ show ax ++ ", " ++ show ay ++ ")"

main = do
  print $ Point { x = 3, y = 3 } `compare` Point { x = 1, y = 2 }
  print $ Point { x = 3, y = 3 }
```

```
GT
(3.0, 3.0)
```

Enum examples.

```
['a'..'e']
succ 'b'
pred 'x'
[LT .. GT]
```

```
abcde
'c'
'w'
[LT,EQ,GT]
```

10.3 Deriving examples

- Haskell in some situations can automatically make our types an instance of the following typeclases: `Eq` `Ord` `Enum` `Bounded` `Show` `Read`
- you can write a lot about them but really what matters are examples

```
data User = User {name:: String, age :: Int} deriving (Eq, Show, Read)

mike = User { name = "Micheal", age = 42 }
john = User { name = "John", age = 23 }

mike == john
mike == User { name = "Micheal", age = 42 }
print mike
read "User {name = \"Micheal\", age = 42}" == mike
```

```
False
True
User name = "Micheal", age = 42
True
```

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
↪ Sunday
  deriving (Eq, Ord, Show, Read, Bounded, Enum)

main = do
  -- because Eq
  print $ Monday == Monday
  print $ Tuesday == Thursday

  -- because Ord
  print $ Friday > Tuesday
  print $ Monday `compare` Sunday

  -- because Show/Read
  print Tuesday
  print (read "Sunday" :: Day)

  -- because Bounded
  print (minBound :: Day)
  print (maxBound :: Day)

  -- because Enum
  print $ succ Monday
  print $ pred Sunday
```

```
  print [Tuesday .. Saturday]
```

```
True
False
True
LT
Tuesday
Sunday
Monday
Sunday
Tuesday
Saturday
[Tuesday,Wednesday,Thursday,Friday,Saturday]
```

10.4 Custom type class

```haskell
data ShirtSize = S | M | L

class MyEq a where
  areEqual :: a -> a -> Bool

instance MyEq ShirtSize where
  areEqual S S = True
  areEqual M M = True
  areEqual L L = True
  areEqual _ _ = False

main = do
  print $ areEqual S M
  print $ areEqual M M
```

```
False
True
```

10.5 Type synonyms

- Those are just aliases to already existing types
- type synonyms can also accept arguments

```haskell
type <new name> = <already existing>
-- for example
type String = [Char]
```

```haskell
type PhoneBook = [(String, String)] -- type
type Name = String -- type
type AssocList k v = [(k, v)] -- type constructor
```

10.6 More advanced

How `Eq` is defined.

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

A typeclass and be a subclass of another typeclass. Here's the first line of definition of `Num`

```haskell
class (Eq a) => Num a where
  ...
```

In order to make a type constructor an instance of a typeclass we can do:

```haskell
instance Eq(m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

10.7 `YesNo` example

We will try to replicate the *true-ish false-ish* values present in Javascript but also in python.

```haskell
class YesNo a where
  yesno :: a -> Bool
```

- we declare a new typeclass called `YesNo`
- it has only want function `yesno`
- now create some instances of that class

```haskell
instance YesNo Int where
  yesno 0 = False
  yesno _ = True

-- this obviously covers strings as well
instance YesNo [a] where
  yesno [] = False
  yesno _ = True

instance YesNo Bool where
  yesno b = b

instance YesNo (Maybe a) where
  yesno Nothing = False
  yesno _ = True
```

Now let's create a `YesNo` counterpart of `if`

```haskell
yesnoIf :: (YesNo a) => a -> b -> b -> b
yesnoIf cond x y = if yesno cond then x else y
```

Let's put this all to work

```haskell

main = do
  print $ yesno (0::Int)
  print $ yesno (-123 :: Int)
  print $ yesno ""
  print $ yesno "hello there"
  print $ yesno (Just 0)
  print $ yesno Nothing
  putStrLn $ yesnoIf (Just 123) "Okay" "Bad"
```

11 Functor

11.1 Basics

- the purpose of `Eq` is to generalize equating things
- the purpose of `Ord` is to generalize comparing things
- in the same spirit a `Functor` is there to generalize mapping over values
- a list is an instance of `Functor` type-class
- types that are instances of `Functor` can usually be thought of as boxes that hold the actual vales in some kind of structure, so instances of Functor include:

  - `[]`
  - `Maybe`
  - `Data.Map`
  - `Tree`
  - `Either` when you keep it mind that `Left` usually represents an error and `Right` the result

- a stricter/better term for functor instead of a box is an computational context: `Maybe` contains some value but also indicates that an operation might have failed. `List` is a undecided value(there are many possibilities).

- If you think of functors as things that output values what `fmap` does really is: attaching a transformation to the output of the functor

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

- notice that `f` is not a placeholder for a concrete type (`Int`, `Char`, `[Float]`...)

- `f` is a type constructor that takes one type as a parameter

- let's compare `fmap` with `map`

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

```
import qualified Data.Map as Map
fmap (*2) [1..3]
map (*2) [1..3]
people = Map.fromList ([(1, "Bob"),(3, "John"),(4, "Tom")])
fmap (++ " Smith") people

fmap (+2) (Just 3)
fmap (+2) Nothing
```

```
[2,4,6]
[2,4,6]
fromList [(1,"Bob Smith"),(3,"John Smith"),(4,"Tom
↪ Smith")]
Just 5
Nothing
```

Some instance of `Functor`.

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

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

instance Functor (Either a) where
  fmap f (Left  x) = Left x
  fmap f (Right x) = Right (f x)

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

11.2 IO functor

- `IO` is an instance of a functor.
- `IO string` can be thought of as a box that goes out into the real world and fetches you a value.
- with `fmap f <io-action>` we can process the content of the IO action using pure/basic functions

```haskell
instance Functor IO where
  fmap f action = do
    value <- action
    return (f value)
```

For example:

```haskell
main = do
  contents <- fmap (takeWhile (/=':') . head . lines) (readFile
  ↪  "/etc/passwd")
```

```haskell
  putStrLn contents
```

```
root
```

11.3 Function functor

- Function are also functors
- the `fmap` is just function composition
- Why this definition? You can think that a for example `(+100)` is a box containing it's eventual value and then it's natural that if we want to change that value in the box the function composition is the way to go
- Say we have a function like `Int -> Char` you can think of it as a large box that contains **every single one of the functions possible outputs**. So in essence it's a collection of values. When we do `fmap t f` we are attaching the `t` transformation to every single one of those values.

```haskell
instance Functor ((->) r) where
 -- fmap :: (a-> b) -> ((->) r a)  -> ((->) r b)
 -- fmap :: (a-> b) -> (r -> a)  -> (r-> b)
    fmap f g = (\x -> f (g x))
```

```haskell
fmap (+3) (*10) $ 2
```

```
23
```

11.4 Lifting a function

Because of the currying behavior of haskell we think of `fmap` in two ways

- the first one is: take a mapping function apply to a *box* and produce a new box with updated values
- the second one is: take a mapping function a produce a mapping between functors

```haskell
fmap :: (a->b) -> f a -> f b
fmap :: (a->b) -> (f a -> f b)
```

For example

```haskell
-- takes a functor over numbers and returns a functor over numbers
:t fmap (*2)
-- takes a functor over strings and returns a functor over strings
:t fmap (++"!")
-- takes a functor over anyting and returns a functor over lists of
↪ anything
:t fmap (replicate 3)
```

```
fmap (*2) :: (Functor f, Num b) => f b -> f b
fmap (++"!") :: Functor f => f [Char] -> f [Char]
fmap (replicate 3) :: Functor f => f a -> f [a]
```

11.5 Functor laws

$$\text{fmap}(\text{id}) = \text{id}$$

$$\text{fmap}(f \circ g) = \text{fmap}(f) \circ \text{fmap}(g)$$

- you can think that those two properties ensure that mapping preserves the structure, the changes are only introduced by the usage of f
- in order to use functors and functions associated with them you need to make sure that those two conditions hold

11.6 Playing around wth functor operators

```haskell
-- fmap :: (Functor f) => (a -> b) -> f a -> f b
fmap (+2) (Left 3)
fmap (+2) (Right 3)

fmap (*2) [1..5]

fmap (+1) (Just 3)
fmap (+1) Nothing

fmap (+1) (0,0)
fmap (+1) (0,0,0)
```

```
Left 3
Right 5
[2,4,6,8,10]
Just 4
Nothing
(0,1)
(0,0,1)
```

```haskell
import Data.Char
-- ($) :: (a -> b) -> a -> b
-- (<$>) :: (Functor f) => (a -> b) -> f a -> f b
(+2) <$> (Right 3)
(+1) <$> (Just 3)


toUpper <$> "Hello world"
(map toUpper) <$> ["hello there", "hello world"]

(+1) <$> (*10) $ 1

(+1) <$> (0, 0)
```

```
Right 5
Just 4
HELLO WORLD
["HELLO THERE","HELLO WORLD"]
11
(0,1)
```

```haskell
-- (<$) :: a -> f b -> f a
1 <$ Left 2
1 <$ Right 3

'a' <$ [1..5]
'a' <$ []

'a' <$ Just 1
'a' <$ Nothing

1 <$ (*10) $ 5

1 <$ (0,0)
```

```haskell
1 <$ (0,0,0)
```

```
Left 2
Right 1
aaaaa

Just 'a'
Nothing
1
(0,1)
(0,0,1)
```

11.7 Make custom list an instance of functor

```haskell
data MyList a = EmptyList | Cons a (MyList a) deriving Show

instance Functor MyList where
  fmap _ EmptyList = EmptyList
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)

main = do
  print $ fmap (*2) (Cons 3 (Cons 4 (Cons 5 EmptyList)))
  print $ 3 <$ (Cons 3 (Cons 4 EmptyList))
```

```
Cons 6 (Cons 8 (Cons 10 EmptyList))
Cons 3 (Cons 3 EmptyList)
```

12 Applicative functors

12.1 Basics

- `fmap` works for functions that take a single argument.
- We want to be able to work with multiparameter functions.

Let's see what happens we try to use binary functions with `fmap`

```haskell
:t fmap (+) (Just 3)
:t fmap compare (Just 8)
:t fmap (++) ["hello", "hi"]
```

```haskell
:t fmap (\x y z -> x*y -z) [3, 4, 8]
```

```
fmap (+) (Just 3) :: Num a => Maybe (a -> a)
fmap compare (Just 8) :: (Ord a, Num a) => Maybe (a ->
↪  Ordering)
fmap (++) ["hello", "hi"] :: [[Char] -> [Char]]
fmap ( y z -> x*y -z) [3, 4, 8] :: Num a => [a -> a -> a]
```

We see that we get functions that are wrapped in functors/boxes/contexts. So in order to be able to work with them further down the line we need to be able to operate/execute such functions that are inside of functors.

This is where the `Applicative` typeclass comes into play:

```haskell
class (Functor f) => Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

- in order for a type constructor to be `Applicative` it needs to be a `Functor`
- `pure` function wraps values inside of default/minimal context
- `<*>` is an inline function that does exactly what we need, meaning it takes a functor that contains a function and a functor over type `a` and produces a functor over type `b`
- `<*>` is really generalized `fmap`

12.2 Maybe

`Maybe` is an instance of `Applicative`

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

- minimal context for `Maybe` is `Just`, it's not `Nothing` because we cannot put any function into it
- if we try to *apply* `Nothing` to something we get nothing
- otherwise we extract the function from `Just f` and apply it to the right side of `<*>`

```
Just (+3) <*> (Just 8)
pure (+) <*> (Just 3) <*> (Just 8)
Nothing <*> (Just 3) <*> (Just 8)
pure (+) <*> Nothing <*> (Just 8)
(:) <$> (Just 3) <*> (Just [4])
```

```
Just 11
Just 11
Nothing
Nothing
Just [3,4]
```

There exist a shorter syntax for `pure (+) <*> ...` Notice the similarity to the normal function application.

```
(++) <$> (Just "hello") <*> (Just " world")
(++)            "hello"            " world"
```

```
Just "hello world"
hello world
```

12.3 List

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

- minimal context is a one element list
- when we want to apply functions from one list to another we create a new list of all possible combinations
- you can think that a list represents a non-deterministic value
- so when we have a non-deterministic function(there are multiple of them) and non-deterministic variable it makes sense to create all of those combinations

```
[(+2), (*3), (subtract 2)] <*> [2,3]
(++) <$> ["hi", "hello", "welcome"] <*> ["!", "."]
(*) <$> [1,2,3] <*> [4,5]
```

```
[4,5,6,9,0,1]
["hi!","hi.","hello!","hello.","welcome!","welcome."]
[4,5,8,10,12,15]
```

12.4 IO

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

```haskell
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b

-- we could rewrite the previous function simply as
myAction = (++) <$> getLine <*> getLine
```

12.5 Function

```haskell
instance Applicative ((->) r) where
  pure x = (\_ -> x)
  f <*> g = (\x -> f x (g x))
```

```haskell
(*) <*> (+3) $ 5
-- (\x -> (*) x (+3 x))
add3 x y z = x + y + z
add3 <*> (+3) <*> (*2) $ 3
-- (\x -> add3 x (x + 3))
-- (\x -> \x -> add3 (x*2) ((x*2) +3))
add3 <$> (+3) <*> (*2) <*> (^2) $ 2
```

```
40
15
13
```

```haskell
-- (*) <$> (+3)
-- 1 -> (* 4)
-- 2 -> (* 5)
-- 3 -> (* 6)

(+) <$> (+3) <*> (*100) $ 5
-- v -> (v+3)
-- v -> (v+3) + _
-- v -> (v+3) + (*100 v)
```

```
508
```

```haskell
(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5

-- v -> (v+3)
-- v -> (\y z -> [v+3, y, z])
-- v -> (\y z -> [v+3, y, z])
-- v -> (\ z  -> [v+3, (*2 v), z])
-- v ->  [v+3, (* 2 v), (_/2)]
-- [5+3, (*2 5), (5/2)]
-- [8, 10, 2.5]
```

```
[8.0,10.0,2.5]
```

12.6 Ziplist

- There are multiple viable implementations of `pure` and `<*>` for lists
- One of them is `ZipList` which can be really useful for example when dealing with mathematical vectors

```haskell
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

```haskell
import Control.Applicative

ZipList [1,2,3,4]
getZipList $ ZipList [1,2,3]

getZipList $ (*2) <$> ZipList [1,2,3]
getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [-1, -2, -3]
```

```haskell
getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [-1, -2, -3]
```

```
ZipList getZipList = [1,2,3,4]
[1,2,3]
[2,4,6]
[0,0,0]
[0,0,0]
```

12.7 Laws

1. `pure f <*> x = fmap f x`
2. `pure id <*> v = v`
3. `pure (.)  <*> u <*> v <*> w = u <*> (v <*> w)`
4. `pure f <*> pure x = pure (f x)`
5. `u <*> pure y = pure ($ y) <*> u`

123