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 Bool

True or False values. Results of comparisons.

```
2 == 0
2 == 2
2 /= 1 -- not equal
2 > 1
2 < 1
2 <= 2
2 >= 3
```

```
False
True
True
True
False
True
False
```

```
True && False
False || True
not False
```

```
False
True
True
```

2.5 Char

Single quotes ''

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

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

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

2.6 Constants

```
mathPi = 3.14159 :: Double
```

3 Math

```
sum [1..5]
mod 5 4
5 `mod` 4 -- infix

pi
sqrt pi
n = 9 :: Int
sqrt (fromIntegral n)
```

```
15
1
1
3.141592653589793
1.7724538509055159
3.0
```

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

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

```
81.0
3.0
2.718281828459045
1.0
10.0
```

```
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
```

4 Lists

4.1 Creating lists

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

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

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

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

```
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]]
```

4.2 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
head nums
last nums
init nums
take 3 nums
drop 3 nums
```

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

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

```
True
13
2
43
20592
```

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

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

```haskell
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]
```

4.3 List comprehensions

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

```haskell
[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]
```

4.4 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
```

4.5 TakeWhile

```haskell
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]
```

5 Strings

Strings are just lists of characters.

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

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

```
ello !
```

6 Tuples

Similar to tuples in python.

```haskell
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

```haskell
isAdult (name, age) = age >= 18
isAdult user
-- 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.

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

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

7 Main function

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

```haskell
import System.IO

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

8 Function

8.1 Basic

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

main = do
  print (fact 5)
```

```
120
```

8.2 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
```

8.3 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]
```

```
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
```

```haskell
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"]
```

8.4 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
```

8.5 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
```

8.6 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
```

8.7 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
```

8.8 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
```

8.9 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)]
```

8.10 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]
```

9 Modules

9.1 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
```

9.2 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

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

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

10 Enumeration types

We create enumeration by using `data` keyword

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

```haskell
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
```

11 Type classes

11.1 Basics

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

We will define Employee.

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

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

```haskell
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
```

```
False
Employee name = "Sam Smith", position = "Manager", idNum = 1
```

- 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
```

11.2 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
```

```haskell
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.

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

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

11.3 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

```haskell
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
```

```haskell
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]
```

11.4 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
```

11.5 Type synonyms

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

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

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

11.6 More advanced

How `Eq` is defined.

```
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`

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

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

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

```
    _ == _ = False
```

11.7 `YesNo` example

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

```
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

```
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`

```
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

```

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

```haskell
  putStrLn $ yesnoIf (Just 123) "Okay" "Bad"
```

```
False
True
False
True
True
False
Okay
```

12 Functor

12.1 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 represent 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

```haskell
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`.

```
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)
```

12.2 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

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

For example:

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

12.3 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.

```
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))
```

12.4 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

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

For example

```
-- 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]
```

12.5 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

13 Applicative functors

13.1 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`

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

```
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`

13.2 Maybe

`Maybe` is an instance of `Applicative`

```
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)
```

```haskell
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.

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

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

13.3 List

```haskell
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

```haskell
[(+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]
```

13.4 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
```

13.5 Function

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

```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]
```

13.6 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

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

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

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

13.7 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
```

14 Monoid

14.1 Formal definition

Set S and a binary operation $S \times S \to S$ (which will be denoted as $\odot$) is a monoid if:

1. the $\odot$ operation is associative

$$\forall a, b \in S \quad (a \odot b) \odot c = a \odot (b \odot c)$$

2. there exist an identity element

$$\exists e \in S \; \forall a \in S \quad e \odot a = a \odot e = a$$

A monoid in which every element has an inverse is a group. Examples of monoids:

1. $S =$ Ints $a \odot b = a \cdot b$, so our elements are integers and our operation is integer multiplication. In this case our identity element $e = 1$
2. $S =$ Set of every possible string $a \odot b = a{+}{+}b$, so our elements are strings and our operation is string concatenation. Identity element $e =$ ""

14.2 In haskell

```haskell
class Monoid m where
  mempty :: m
  mappend : m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

1. mempty $= e$
2. mappend $= \odot$
3. `mconcat` function exists because there could a more efficient way to implement it than the default

14.3 Instances of monoid

14.3.1 Lists

```haskell
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

```haskell
[1,2,3] `mappend` [4,5,6]
"one " `mappend` "two" `mappend` "three"
mconcat ["hello", " there", "!"]
```

```
[1,2,3,4,5,6]
one twothree
hello there!
```

14.3.2 Numbers

There are multiple monoids when it comes to numbers. Here is the monoid in which our operation is a multiplication.

```
import Data.Monoid
getProduct $ Product 3 `mappend` Product 9
getProduct $ Product 3 `mappend` mempty
getProduct $ mconcat [Product 3, Product 4, Product 5]
(getProduct . mconcat . map Product) [3,4,2]
```

```
27
3
60
24
```

There also exist `Sum`

```
import Data.Monoid
getSum $ Sum 2 `mappend` Sum 9
getSum $ mempty `mappend` Sum 9
(getSum . mconcat . map Sum) [1,2,3]
```

```
11
9
6
```

14.3.3 Bools

There exist `Any` and `All` in `Data.Monoid`, `Any` is connected with logical or and `All` is logical and. You use them in the exact same way as `Sum` and `Product`.

14.3.4 Ordering

```
instance Monoid Ordering where
  mempty = EQ
  LT `mappend` _ = LT
  EQ `mappend` y = y
  GT `mappend` _ = GT
```

1. identity element = `EQ`
2. We keep the left value unless it's equal to `EQ` in which case we choose the right one. This resembles the way we compare words.

Let's say we want to create a function that compares two strings base on their length in the case of lengths being equal we wan to compare the alphabetically.

```
lenCmp :: String -> String -> Ordering
lenCmp x y =
  let a = length x `compare` length y
      b = x `compare` y
  in  if a == EQ then b else a
```

By using our monoid we can rewrite it:

```
import Data.Monoid
lenCmp :: String -> String -> Ordering
lenCmp x y = (length x `compare` length y) `mappend` (x `compare` y)
```

If we wanted to add another criterion it's really easy. <> is the same as `mappend`

```
import  Data.Monoid

fancyCompare :: String -> String -> Ordering
fancyCompare x y =
  (length x `compare` length y) -- most imporant criterion
    <> (vowels x `compare` vowels y)
    <> (special x `compare` special y)
    <> (x `compare` y) -- least important criterion

 where
  vowels  = length . filter (`elem` "aeiou")
  special = length . filter (`elem` "!@#$%^&*()")

main = do
  print $ fancyCompare "bbbb" "zzz"
  print $ fancyCompare "aabc" "bibc"
  print $ fancyCompare "aab&" "bibe"
  print $ fancyCompare "aab&" "bi*e"
```

```
GT
GT
GT
LT
```

14.4 `Foldable`

- `Foldable` type-class existing to generalize turning a collection of values(list, tree, queue, heap, etc.) into a single value

- using this type-class for example finding maximum/minimum/average values can be generalized for pretty much any data structure
- two types of folds:
  - lazy - work with infinite lists, create lots of thunks which can be detrimental to performance
  - strict - force each step of the fold to be reduce to Weak Head Normal Form, great for reducing finite structures to a single number(for example sum)
- In order to perform a reduction we use `Monoids`

In order create a instance of `Foldable` we need to implement `foldMap` or `foldr`.

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Eq)

instance Foldable Tree where
  -- foldMap :: Monoid m => (a->m) -> Tree a -> m
  foldMap f Empty                = mempty
  foldMap f (Node v left right) = foldMap f left <> f v <> foldMap f right

tree :: Tree Int
tree = Node 1 (Node 2 Empty Empty) (Node 3 Empty (Node 5 Empty Empty))

main = do
  print $ foldl (+) 0 tree
  print $ foldMap (\x -> [x]) tree
  print $ sum tree
  print $ product tree
  print $ maximum tree
  print $ minimum tree
```

```
11
[2,1,3,5]
11
30
5
1
```

15 Monad

15.1 How to discover it?

The following are my notes from Monads I by Graham Hutton.

Example: a simple evaluator for expressions consisting of either: numbers or division of numbers.

```
data Expr = Val Int | Div Expr Expr
```

We then want to create a function which will evaluate such expressions.

```
eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = eval x `div` eval y
```

The problem is: this program crashes when division by zero happens. We can we `Maybe` in order to indicate failure.

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)
```

No we can rewrite our evaluator.

```
eval :: Expr -> Maybe Int
eval (Val n  ) = Just n
eval (Div x y) = case eval x of
  Nothing -> Nothing
  Just n  -> case eval y of
    Nothing -> Nothing
    Just m  -> n `safeDiv` m
```

This will never crash but it's quite ugly. We can try to rewrite it using `Applicative`.

```
eval :: Expr -> Maybe Int
eval (Val n  ) = pure n
eval (Div x y) = safeDiv <$> (eval x) <*> (eval y)
```

Even thought it's a lot simpler it does not work. Because of a type error. In the third line the resulting type is actually `Maybe (Maybe Int)`. It would work if `safeDiv` had the type of `Int -> Int -> Int` but then we could not signify failure. **Applicative style is useful when we want to apply pure functions to impure functors**.

Monad is a common pattern that we can be abstracted away.

```
case mx of
  Nothing -> Nothing
  Just x -> f x
```

Where `mx` is some value and `f` is a function.

```
(>>=) :: (Maybe a) -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
        Nothing -> Nothing
        Just x -> f x
```

We abstracted away then common pattern of case analysis. `>>=` is often called into or bind.

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x >>= (\n ->
                 eval y >>= (\m ->
                 safeDiv n m))
```

The general pattern. `m1,m2,...` are `Maybe` values.

```
m1 >>= \x1 ->
m2 >>= \x2 ->
  .
  .
  .
mn >>= \xn ->
  f x1 x2 ... xn
```

This pattern is so command that haskell has special syntax for it called the `do` notation. The previous code is equivalent to:

```
do x1 <- m1
   x2 <- m2
      .
      .
      .
   xn <- mn
   f x1 x2 ... xn
```

The following is the `eval` function written using the `do` notation.

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safeDiv n m
```

This is as simple as the original `eval` definition which failed when division by 0 happened. It also looks like an imperative program.

15.2 Formalities and examples

15.2.1 Definition in haskell

The following are my notes from Monads II by Graham Hutton.

In Haskell the class of applicative functors that support the bind operator(>>=)are called monads.

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a    -- like pure
  return = pure         -- default implementation
```

15.2.2 Maybe Monad

Let's create a `Maybe Monad`.

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

15.2.3 List Monad

```
instance Monad [] where
  -- (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= f = concat $ map f xs
           = [y | x <- xs,  y <- f x] -- equivalent to previous
```

Let's say we want to create a function `pairs` which will produce essentially an cartesian product of two lists.

```
pairs :: [a] -> [b] -> [(a,b)]
                  -- chose one value from xs in all possible ways
pairs xs ys = do x <- xs
                  -- chose one value from ys in all possible ways
                  y <- ys
                  -- produce the pair
                  return (x,y)
main = do
  print $ pairs [1,2] [8,9]
```

```
[(1,8),(1,9),(2,8),(2,9)]
```

Meanings:

- `x <- xs` choose one value from `xs` in all possible ways

- `x <- xs` choose one value from `ys` in all possible ways
- `(x,y)` produce the pair

15.2.4 State Monad

```
type State = ...
type ST a = State -> (a, State)
```

Worth stating explicitly:

- `State` can be really anything
- `ST` stands for state transformer. `a` is some kind of result. `State` is possibly modified state.

Let's image we have a function `Char -> ST Int` this is equivalent to `Char -> State -> (Int, State)`. This is a curried function which takes a `Char` and `State` as input.

Let's see how this all comes down to create a monad. In the following piece of code `newtype` is used in improve performance.

```
type State = Int
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) = st -- apply the state transformer

instance Functor ST where
  -- fmap :: (a->b) -> ST a -> ST b
  fmap f st =
    S (\state -> let (a, newstate) = app st state in (f a, newstate))

instance Applicative ST where
  -- pure :: a -> ST a
  pure x = S (\s -> (x, s))

  -- (<*>) :: ST (a->b) -> ST a -> ST b
  first <*> second = S
    (\state ->
      let (f, new_state) = app first state in app (fmap f second)
      ↪  new_state
    )

instance Monad ST where
  -- return :: a -> ST a
  return x = S (\s -> (x, s))
```

```haskell
  -- (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f = S (\s -> let (x, s') = app st s in app (f x) s')
```

16 newtype

- It's used simply for wrapping an existing type, for that we can for example redefine it's implementation of `Applicative` / `Functor` typeclasses.
- Why not use `data` keyword? Because `newtype` was made with specifically the wrapping purpose in mind. Thus Haskell can optimize it's usage.

```haskell
newtype Zp a = Zp {getZp :: [a]}

instance Functor Zp where
  fmap f (Zp xs) = Zp (fmap f xs)


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

main = do
  print $ getZp $ (+) <$> Zp [1,2,3] <*> Zp [4,5,6]
```

```
[5,7,9]
```

17 Types

17.1 Basic example

We can use the `data` keyword to create our own data types.

For example the `Bool` data type is defined as follows:

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

- The `False` and `True` are value constructors.
- The `Bool` is the type constructor

We can create a more elaborate data type.

- The circle consists of: x, y of center and radius

- The rectangle: x y of lower left, x y of upper right

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
↪  deriving (Show)
:t Circle
:t Rectangle

originCircle = Circle 0 0

map originCircle [2, 3, 5]
```

```
Circle :: Float -> Float -> Float -> Shape
Rectangle :: Float -> Float -> Float -> Float -> Shape
[Circle 0.0 0.0 2.0,Circle 0.0 0.0 3.0,Circle 0.0 0.0 5.0]
```

Value constructors are functions thus we can use nice features like partial application etc with them.

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
↪  deriving (Show)

surface :: Shape -> Float
surface (Circle _ _ r         ) = pi * r * r
surface (Rectangle x1 y1 x2 y2) = abs (x2 - x1) * abs (y2 - y1)

main = do
  print $ surface $ Circle 2 3 1
  print $ surface $ Rectangle 0 0 3 4
```

```
3.1415927
12.0
```

17.2 Types and modules

To create a module with everything that was created in the shapes example we could write:

```
module Shapes(Shape(..), surface, originCircle) where
```

- The `Shape(..)` means we are exporting every value constructor for that type
- You can also directly specify which value constructors you want to export by using `Shape(Rectangle, Circle)`

- Also you can write just `Shape`, in that case no value constructor will be exported. `Data.Map` uses this approach. Then you need to create appropriate functions that create a value of that type and expose them. This gives you **full** control of how values of your data type are created.

17.3 Record Syntax

- Record syntax is meant to be used when attributes of a value constructor have an arbitrary order. So for example
- nicer string representation
- it automatically creates functions to access various attributes

```
data Car = Car
  { make    :: String
  , model   :: String
  , mileage :: Int
  }
  deriving Show

car = Car {make = "Model T", model = "Ford", mileage = 1000}

main = do
  putStrLn $ make car
  putStrLn $ model car
  print $ mileage car
  print $ car
```

```
Model T
Ford
1000
Car make = "Model T", model = "Ford", mileage = 1000
```

17.4 Type parameters

17.4.1 General Info

- very similar to generics
- **value** constructors take some **values** as arguments and create a new value
- in a similar way **type** constructors take **types** as arguments and create a new type

- type parameters are especially useful when we are creating a type that acts as a box. Those include things like ADTs or the `Maybe`.

An example is `Maybe` type constructor from the standard library.

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

```
Just "Hello"
Just 84
:t Just "Haha"
:t Nothing
```

```
Just "Hello"
Just 84
Just "Haha" :: Maybe String
Nothing :: Maybe a
```

- Notice that the type of `Nothing` is polymorphic meaning if a function accepts values `Maybe a` we can always pass `Nothing` to it.
- Also list is a type constructor but with some syntactic sugar. The `[]` has a similar function and properties as `Nothing` from the example.

If we were designing our own mapping type we might be tempted to do the following:

```
data (Ord k) => Map kv  = ...
```

Since in order to built a tree we need ordering. Theoretically it makes sense. But in practice we **should never do that**. The reason is that we will have to use the `Ord` type-class in every single function that deals with maps. Even in those that do not depend on the ordered nature of the keys. Thus we will create unnecessary clutter.

17.4.2 Vector example

We will create a 3D vector type that will be able to hold any kind of number you want (Int, Double, ...).

```
data Vector a = Vector a a a deriving Show

vplus :: (Num t) => Vector t -> Vector t -> Vector t
(Vector x y z)  `vplus` (Vector a b c) = Vector (x + a) (y + b) (z + c)

vdotprod :: (Num t) => Vector t -> Vector t -> t
```

```
(Vector x y z)  `vdotprod` (Vector a b c) = (x * a) +  (y * b) + (z * c)

main = do
  print $ Vector 1 2 3 `vplus` Vector 0 1 4
  print $ Vector 1.2 3.1 4.2 `vdotprod` Vector 0.0 1.0 2.0
```

```
Vector 1 3 7
11.5
```

17.5 Either

- it is somewhat similar to `Maybe`
- we use `Either` if our computation can fail in more ways than 1
- typically then we use something like `Either String <ValueType>` where `Left` represents an error message and `Right` contains the actual value
- so if you get a `Left` value something went wrong
- a `Right` value indicates that everything in theory was okay

`Either` type constructor is roughly defined:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Let's see it in action.

```
Right 20
Left "hello"
:t Right 'a'
:t Left True
```

```
Right 20
Left "hello"
Right 'a' :: Either a Char
Left True :: Either Bool b
```

17.6 Recursive data types

- `[1]` is a syntactic sugar for `1:  []`
- for `[1,2,3]` it's `1:2:3:[]`
- we can say that a list is either an empty list or it's an element joined with another list

- notice the recursive nature of that definition

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
Empty
Cons 2 Empty
Cons 1 $ Cons 2 $ Cons 3 $ Cons 4 Empty
```

```
Empty
Cons 2 Empty
Cons 1 (Cons 2 (Cons 3 (Cons 4 Empty)))
```

Quick side note:

- we can make functions/type constructors infix by default by only composing them of special characters
- fixty declarations allows to specify if our operator is left or right associative and how tightly it bounds
- for example `infixl 7 *` and `infixl 6 +` means that `*` and `+` are left associative and that `*` bounds more tightly than `+`
- `infixr 5 :-:` means that our operator is right associative and it binds less tightly than for example `+`

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)

main = do
  print (Empty :: List Int)
  print (2 :-: 3 :-: 5 :-: Empty)
```

```
Empty
2 :-: (3 :-: (5 :-: Empty))
```

17.7 BST trees

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Eq,
↪  Read)

leafNode :: a -> Tree a
leafNode x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = leafNode x
treeInsert x (Node y left right) | x == y = Node x left right
```

```
                                | x > y  = Node y left (treeInsert x
                                ↪  right)
                                | x < y  = Node y (treeInsert x left)
                                ↪  right

treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node y left right) | x == y = True
                               | x > y  = treeElem x right
                               | x < y  = treeElem x left

tree :: Tree Int
tree = foldr treeInsert EmptyTree [1,2,4,3]

main = do
  print $  4 `treeElem` tree
  print $  2 `treeElem` tree
  print $  123 `treeElem` tree
```

```
True
True
False
```

18 IO

18.1 Hello World

The typical hello world program in haskell, which we save in `helloworld.hs`

```
main = putStrLn "Hello, world!"
```

To create an executable and run it we do:

```
ghc --make helloworld
./helloworld
```

18.2 IO actions

Let's take a closer look at `putStrLn`.

```
:t putStrLn
```

`putStrLn :: String -> IO ()`

- This functions accepts a string and returns `IO ()`.

- In Haskell we want to have as much of pure code as possible.
- A pure function has a very strict property: **given the same arguments a pure function will always return the same result**
- This property makes pure functions/code very easy to test/write/understand.
- However a pure function by definition cannot interact with the file system/network. Because then the function output may vary(for example: file may exist or not).
- **IO action can interact with and change the world outside our program**

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

- `IO a` type represents an action that when called will carry out a side-effect and always come back with a value, but that value may be just an empty tuple.
- IO action can be thought of as a *box with legs*. We send it to perform a side-effect and in the meantime it will collect some data usually related to the performed side-effect.
- When are IO actions executed? When we run our haskell program and the `main` IO action is called. Then all of the other functions that return `IO <whatever>` will be called from this main function or from the function that was called in the main function and so on.
- `main` acts like a sink for all of our IO actions. It it the start of our *impure* part of code.
- To be able to run multiple IO actions in `main` we can use the `do` syntax. Which is actually just a syntactic sugar for working with monads. That will be covered later. For now it's just a way of gluing together a bunch of IO actions.

```
main = do
  putStrLn "Enter your name:"
  name <- getLine
  putStrLn $ "Nice to meet you " ++ name
```

- Compile it, run it and it will behave how you might expect.

- This piece of code looks very much like an imperative program.
- The second line gets input from the user and it appears as if it was stored in a *variable* named `name`. What we actually would is that the `<-` operator binds the result of IO action `getLine` to `name`.
- `getLine` returns `IO String` but since we are using the `<-` operator `name` will actually have `String` type.
- You can use `<-` (taking data out of an IO action) **only in another IO action.** This creates a nice separation between our pure and impure code.

```
:t getLine
```

```
getLine :: IO String
```

- `getLine` is a function that accepts no arguments and returns `IO string`. Meaning it will perform a side-effect and come back to us with a string value.

18.3 Why is IO in Haskell considered pure?

18.3.1 Real world is a value

If we think of a function like `putStrLn` as a function that takes two parameters:

- state of the real world
- string to write

and then returns the new state of the real world. This function is pure. Simply because we cannot pass the same argument to it twice.

My own personal opinion: **this is vague**. I am aware that idea is actually present in the `ghc`. However it is not a convincing argument. If such a function would be considered pure by us then we can make **every single function** pure. There is obviously a problem with parallel/concurrent execution. *Actually pure* functions work really well in that regard and have some clever optimizations that only work with such functions. But what about our IO? This discussion could lead us to theory of parallel universes! Though this argument is interesting it's filled with paradoxes and ambiguities.

18.3.2 IO action is not pure it's indicates impurity

If we see something like `IO string` in our code this means **impurity**. We should strive to put as much of our code into pure sections of it(not tainted with IO stuff).

My own opinion: **this view is really useful and pragmatic**. I think it's probably the most useful. Saying that Haskell is not entirely pure language to some people is a blasphemy. However when we are actually writing programs I believe that this very idea it of the most value.

18.3.3 Discussion of Haskell's IO

Discussion of Haskell's IO

```
func puts(s string, w world) world

func gets(w world) (string, world)

func main(w world) world
      w1 := puts("Enter a string:", w)
      s, w2 := gets(w1)
      return puts("You typed: " + s, w2)


// after adding a layer of indirection
type IO       func(w world) (        world)
type IOString func(w world) (string, world)
type IOInt    func(w world) (   int, world)
...

func puts(s string) IO
func gets() IOString

func main(w world) world
      s, w1 := gets() (puts("Enter a string:") (w))
      return puts("You typed: " + s) (w1)
```

This code is pure. Everything is immutable. Variable assignments only occur during initialization. I/O is no problem.

How does the compiler deal with the world type? Easy! The compiler initially treats it like any other type, then after applying optimizations that work only for pure functions, it discards objects of type world.

18.3.4 Recipe

If a function returns `IO string` it actually returns just a recipe for how to get the string. When we combine two or more functions that return `IO actions`.

```
getSecondLine
getSecondLine = do
```

18.4 Dealing with `stdin/stdout`

18.4.1 `putStrLn getLine` and `let`

```
import Data.Char

main = do
  putStrLn "Enter first name"
  fname <- getLine
  putStrLn "Enter last name"
  lname <- getLine
  let bigFname = map toUpper fname
      bigLname = map toUpper lname
  putStrLn $ "Hello there " ++ bigFname ++ " " ++ bigLname ++ " !"
```

- Notice that we can use `let` in `do` syntax just like in functions. However we do not write `in` keyword. Notice
- `let` is used to bind names to values from *normal* functions
- `<-` is used to bind names to values from `IO` actions

18.4.2 Continuous input/output and `if`

```
-- reads a line and outputs it with words reversed
main = do
  line <- getLine
  if null line
    then return ()
    else do
      putStrLn $ (unwords . reverse . words) line
      main
```

- Notice the usage of if in the `do` statement.

- `if` inside of a `do` must return an IO action that's why I wrote `else do` instead of just `else`
- We can use recursion just like in any other function

18.4.3 `when/unless`

```
import Control.Monad

main = do
  line <- getLine
  unless (null line) $ do
    (putStrLn . unwords . reverse . words) line
    main
```

- Pretty much self explanatory
- In unless if condition is False we execute the passed IO action, if its True then we return ()

18.4.4 `sequence mapM mapM_ forM forM_`

```
import Data.List

main = do
  ls <- sequence [getLine, getLine, getLine]
  (putStrLn . intercalate "\nBREAK\n") ls
```

- Sequence takes a list of IO actions and produces another IO action that will

execute all of the passed actions and give you a list of results.

```
main = do
  mapM_ print [1, 2, 3, 4]
```

```
1
2
3
4
```

- Map takes an array of values and applies to them an IO action, thus creating a new IO action. `mapM_` does not return the results of IO actions whereas `mapM` does.

```haskell
import Control.Monad

main = do
  forM_
    [1, 2, 3, 4]
    (\a -> do
      putStrLn $ show a ++ ". "
    )
```

```
1.
2.
3.
4.
```

- `forM` is basically just `mapM` but the parameters are reversed. `forM_` does not the returns the

values of computed `IO` actions and `forM` does

18.5 `forever`

```haskell
import Data.Char
import Control.Monad

main = do
 forever $ do
   line <- getLine
   putStrLn $ map toUpper line
```

18.6 Lazy IO with `getContents`

- `getContents` is a really cool function that does Lazy IO.
- This means no data will be fetched until it is needed.
- You can just think that you are transforming the contents of the entire file and let the laziness of Haskell and IO do the buffering.
- You can pipe `cat ~/.bashrc` into the following program and it will give you your aliases

```haskell
import Control.Monad
import Data.List.Split (splitOn)

main = do
  contents <- getContents
```

```
  forM_
    (getAliases contents)
    ( \(name, value) -> do
        putStrLn $ "alias: " ++ name ++ " value: " ++ value
    )

getAliases content =
  aliases
  where
    aliases = map (parseAliasLine . (!! 1)) aliasLines
    aliasLines = filter (\line -> length line == 2 && head line ==
    ↪  "alias") filelines
    filelines = (map words . lines) content

parseAliasLine :: String -> (String, String)
parseAliasLine line = (name, value) where [name, value] = splitOn "=" line
```

18.7 interact

- `interact` is used to transform the passed input String into output String
- Since the IO is lazy you can just think that you are given the whole file as a one huge String and you need to provide output in another String.
- This program counts the number of lines which are palindromes

```
import Data.List.Split (splitOn)
import System.IO (isEOF)

main = do
  interact processContents

processContents :: String -> String
processContents contents =
  createStatsMessage (areLinesPalindromes contents)

createStatsMessage :: [Bool] -> String
createStatsMessage isPalindrome =
  "Palindromes found: " ++ nPalindrome ++ " Total: " ++ n
  where
    n = (show . length) isPalindrome
    nPalindrome = (show . length . filter (== True)) isPalindrome

areLinesPalindromes :: String -> [Bool]
areLinesPalindromes content =
```

```
  let allLines = lines content
      isPalindrome = map (\line -> reverse line == line) allLines
   in isPalindrome
```

This is `text.txt`.

```
ala
abc
aabbaa
asdfasfdsa
89998
```

18.8 Working with files

18.8.1 Basics

- `stdin/stdout` are practically just files
- When we want deal with files other than those two, we need to obtain what's called a handle. Then it's simply a matter of using the same functions as previously but prefixing them with `h`. Obviously we need to pass the file handle to them.
- It's also important to close the file after we are done working with it

```
import System.IO

main = do
  handle <- openFile "/etc/passwd" ReadMode
  contents <- hGetContents handle
  let nLines = length $ lines contents
  putStrLn $ "There are " ++ show nLines ++ " users on this machine"
  hClose handle
```

```
There are 34 users on this machine
```

18.8.2 `withFile`

- `withFile` function opens a file in specified mode and then executed a function which we pass
- **it closes the file automatically** so we don't forget
- this is the previous example but now we use `withFile`

```
import           System.IO

main = do
  withFile
    "/etc/passwd"
    ReadMode
    (\handle -> do
      contents <- hGetContents handle
      let nLines = length $ lines contents
      putStrLn $ "There are " ++ show nLines ++ " users on this machine"
    )
```

```
There are 34 users on this machine
```

18.8.3 `readFile`

- we pass it a file name and we get it's contents as a string
- it's lazy IO

```
main = do
  contents <- readFile "/etc/passwd"
  putStrLn $ (head . lines) contents
```

```
root:x:0:0::/root:/bin/bash
```

18.8.4 `writeFile appendFile`

- you can a file name and a string
- in case of `writeFile` our string **replaces** what was in that file
- in case of `appendFile` our string **is appended** to the end of the file

```
main = do
  writeFile "/tmp/sthsth.txt" "hello world!"
  message <- readFile "/tmp/sthsth.txt"
  putStrLn message
```

```
hello world!
```

18.8.5 Lazy IO

- the default buffering of a file is line-buffering
- you can change that by using `hSetBuffering`

- the following piece of code changes the chunk size to 2048 bytes

```
hSetBuffering handle $ BlockBuffering (Just 2048)
```

18.9 Command line arguments

- `getArgs` gives the passed arguments
- `getProgName` gives the program name

```
import           Control.Monad
import           System.Environment

main = do
  args     <- getArgs
  progName <- getProgName
  forM_
    (zip [1 .. ] args)
    (\(i, arg) -> do
      putStrLn $ show i ++ ".  " ++ arg
    )
  putStrLn $ "program Name: " ++ progName
```

```
# shell
ghc cmd-args.hs
./cmd-args hello there 123 321 -v 1 -u 4
```

19 Random

```
:t random
:t mkStdGen
```

```
random :: (Random a, RandomGen g) => g -> (a, g)
mkStdGen :: Int -> StdGen
```

- `random` function takes a random value generator and outputs a *random* value and a new generator
- random generator is for example: Mersenne twister, so in essence it is a function that takes a value and returns a new one that usually looks very different from the one that was passed in. So in the random generator there ironically nothing random. The randomness comes from a more or less random seed(for example some combination of current times, temperatures etc)

- `mkStdGen` takes a seed and returns a `RandomGenj`

```
import System.Random
random (mkStdGen 10) :: (Int, StdGen)
random (mkStdGen 10) :: (Bool, StdGen)
random (mkStdGen 10) :: (Double, StdGen)
```

```
(3575835729477015470,StdGen unStdGen = SMGen 7847668597276082904
↪  614480483733483467)
(False,StdGen unStdGen = SMGen 7847668597276082904 614480483733483467)
(0.8061535566824866,StdGen unStdGen = SMGen 7847668597276082904
↪  614480483733483467)
```

- **Notice that this is completely deterministic**
- For `Int` the `random` functions returns a random value from `minBound` to `maxBound`
- For `Float` / `Double` it returns a random value from the interval $[0, 1]$
- To generate a couple of random values we need to run `random` with a generator then take newly produced generator and run `random` with it and so

```
import           System.Random

:{
threeCoinsFlips :: StdGen -> (Bool, Bool, Bool)
threeCoinsFlips gen =
  let (first, gen2) = random gen :: (Bool, StdGen)
      (second, gen3) = random gen2 :: (Bool, StdGen)
      (third, gen4) = random gen3 :: (Bool, StdGen)
  in
  (first, second, third)
:}

print $ threeCoinsFlips $ mkStdGen 30
```

```
(False,True,True)
```

We can use `randoms` to generate as many random variables as needed.

```
import System.Random

take 10 $ randoms (mkStdGen 10) :: [Bool]
take 10 $ randoms (mkStdGen 10) :: [Char]
```

```
[False,True,True,True,False,True,False,True,False,True]
5108200782985349139322061703995105610042586638828O5
```

`randomR` and `randomRs` allow us to get random values from a specified interval.

```
import           System.Random

randomR ('a', 'z') (mkStdGen 10)
take 10 $ randomRs ('a', 'z') (mkStdGen 10)
```

```
('m',StdGen unStdGen = SMGen 7847668597276082904 614480483733483467)
mqqxmgkzss
```

`getStdGen` is an IO action that gives you a global random number generator provided by your operating system. Note that if you run this function twice it will give you the same generator.

```
import System.Random
:{
main = do
    gen <- getStdGen
    print $ take 10 $ randomRs (0, 10) gen
:}

main
```

```
[8,6,5,3,2,10,6,0,5,7]
```

`newStdGen` splits the current global generator updates it and gives you the new generator. Not only you get a generator but also the global one is changed.

20 Byte strings

20.1 Why?

Lists are very useful and there are a lot of functions that work with them. The problem is that they are lazy. This often is actually a benefit but the laziness can very negatively effect performance when working with big files and manipulating them.

Byte strings to the rescue. They are byte vectors. Meaning you can think of them as lists of bytes.

There are two types of byte strings. Firstly `Data.ByteString` has no laziness at all. They offer a good speed boast compared to lists(when working with big files). However they can also be very memory intensive. Secondly we have `Data.ByteString.Lazy`. Those byte strings are lazy but not as much of lists. Byte string is divided into chunks(64K each) and those chunks are lazily evaluated. Essentially is like a mixture of lists and `Data.ByteString`.

20.2 Usage

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
abc = B.pack [97, 98, 99]
B.pack [97..122]
B.unpack abc
```

`fromChunks` takes strict byte strings are returns a lazy byte string. `toChunks` takes a lazy byte string and returns strict ones.

```
B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], S.pack [46,47,48]]
```

```
()*+,-./0
```

`cons` works like `:`. `cons'` will try to fit the passed byte into the first chunk if there is space.

```
B.toChunks $ B.cons  122 abc
B.toChunks $ B.cons' 122 abc

B.toChunks $ B.foldr B.cons B.empty $ B.pack [97,98,99, 100, 110]
B.toChunks $ B.foldr B.cons' B.empty $ B.pack [97,98,99, 100, 110]
```

```
["z","abc"]
["zabc"]
["a","b","c","d","n"]
["abcdn"]
```

There is also `readFile :: FilePath -> IO ByteString`. If you use strict byte strings it will read the entire file into memory at once.

This program concatenates multiple files into one. I am assuming that user specified correct file names and has appropriate permissions.

```
import           Control.Monad
import           System.Environment
import qualified Data.ByteString as B
```

```haskell
main = do
  filenames <- getArgs
  concatFiles filenames

concatFiles :: [FilePath] -> IO ()
concatFiles fileNames = do
  forM_
    fileNames
    (\fn -> do
        bs <- B.readFile fn
        B.putStr bs
    )
```

```
echo hello there > 1.txt
echo -e "a\nb\nc" > 2.txt
echo "message!" > 3.txt
runhaskell concat-files.hs 1.txt 2.txt 3.txt
```

```
hello there
a
b
c
message!
```

21 Exceptions

- When an operation can fail we have two main ways of communicating that an error happend:
  - `Maybe` / `Either` types
  - exceptions
- In pure code you should pretty much always favor using `Maybe` / `Either`.
- In IO however exceptions can be useful because otherwise every single IO operation would need to return a `Maybe` / `Either` type and we would need to check if everything was ok with every operation.
- With the help of exceptions we can write our IO code `Maybe` / `Either` free and then enclose those parts with proper exception handling
- Exceptions can **only** be caught in IO part of the code

```haskell
import Control.Exception
import System.Environment
import System.IO.Error

handler :: IOError -> IO ()
handler e
  | isDoesNotExistError e = putStrLn "file does not exist"
  | isPermissionError e = putStrLn "you don't have permission to read this
  ↪  file"
  | otherwise = ioError e

tryGetLines :: IO ()
tryGetLines = do
  (fn : _) <- getArgs
  contents <- readFile fn
  print $ fn ++ " has " ++ (show . length . lines) contents ++ " lines"

main = tryGetLines `catch` handler
```

```
runhaskell exceptionExample.hs ~/.bashrc
runhaskell exceptionExample.hs ~/doesnotexist.txt
runhaskell exceptionExample.hs /etc/shadow
```

```
"/home/piotr/.bashrc has 265 lines"
file does not exist
you don't have permission to read this file
```

22 Kinds

- When we have a function with type signature `f::  a -> b -> b` that means it takes two values of possibly different types and then returns a value whose type is the same as the second argument's.
- If we about for example `Data.Map`, it's type constructor takes two types. We can partially apply a type for the keys so that this new type constructor takes only one type. If we call it again it will now produce a concrete type.
- There is a similarity between those two.
- Types are labels that values carry so that we can reason about the values
- But types have their own labels called **kinds**
- **We can check out the kind of a type by using** `:k <type>`

  - `*` means a concrete type
  - `*->*->*` means it takes two concrete types and produce a concrete type

```
import qualified Data.Map as M

:k Int
:k Maybe
:k M.Map
:k M.Map Int
:k Either
```

```
Int :: *
Maybe :: * -> *
M.Map :: * -> * -> *
M.Map Int :: * -> *
Either :: * -> * -> *
```

- We now have new vocabulary to describe types
- we can now say that `Functor` typeclass expects a type that is of `*->*` kind
- so only types types of kind `*->*` can be be instances of `Functor`

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

23 Useful functions/modules

```
import Data.List
intersperse '.' "Hello"
intercalate " " ["hello", "world", "!"]
intercalate [0,0] [[1,2,3], [4,5,6], [7,8,9]]
```

```
H.e.l.l.o
hello world !
[1,2,3,0,0,4,5,6,0,0,7,8,9]
[[1,5],[2,6],[3,7],[4,8]]
HelloWorld!
[1,2,3,4,5]
[4,5,2,3,4,5,3,4,5]
```

```haskell
import Data.List

transpose [[1,2,3,4], [5,6,7,8]]
concat ["Hello", "World", "!"]

concat [[1,2,3], [4,5]]

concatMap (\x -> [x..5]) [4, 2, 3]
```

```
[[1,5],[2,6],[3,7],[4,8]]
HelloWorld!
[1,2,3,4,5]
[4,5,2,3,4,5,3,4,5]
```

```haskell
and [True, True, True]
and [True, False, True]
or [True, False, False]
or [False, False, False]

and $ map  (>3) [4,9,5]
or $ map  (>=9) [4,9,5]
```

```
True
False
True
False
True
True
```

```haskell
any (>3) [1,2,3,4]
all (>=0) [0,1,2,3]
```

```
True
True
```

24 Association lists

24.1 Normal lists & `lookup`

```haskell
addresses =
  [ ("Bob", "Lipowa 2")
  , ("Tom", "Ruska 8")
  , ("Tim", "Reymonta 9")
  , ("Joe", "Mickiewicza 16")
  ]
```

```haskell
findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key = foldr (\(k, v) acc -> if key == k then Just v else acc)
↪  Nothing

main = do
  print addresses
  print $ findKey "Bob" addresses
  print $ findKey "John" addresses
  print $ lookup "Bob" addresses
  print $ lookup "John" addresses
```

```
[("Bob","Lipowa 2"),("Tom","Ruska 8"),("Tim","Reymonta
↪  9"),("Joe","Mickiewicza 16")]
Just "Lipowa 2"
Nothing
Just "Lipowa 2"
Nothing
```

24.2 Data.Map

Association lists using default lists in haskell are obviously really slow. Usually you want to use `Data.Map` which provides a much more efficient implementation, which internally uses trees, so the keys not only have to be `Eq` but also `Ord`.

```haskell
import qualified Data.Map                    as Map

m = Map.fromList
  [ ("Bob", "Lipowa 2")
  , ("Tom", "Ruska 8")
  , ("Tim", "Reymonta 9")
  ]

main = do
  print m
  print $ Map.lookup "Tom" m
  print $ Map.lookup "Joe" m
  print $ Map.size m
  print $ Map.null m
  print $ Map.insert "Joe" "Kwiatowa 3" m
  print $ Map.member "Tom" m
  print $ Map.map (takeWhile (/=' ')) m
  print $ Map.filter ((=='R') . head) m
  print $ Map.toList m
```

```
fromList [("Bob","Lipowa 2"),("Tim","Reymonta 9"),("Tom","Ruska 8")]
Just "Ruska 8"
Nothing
3
False
fromList [("Bob","Lipowa 2"),("Joe","Kwiatowa 3"),("Tim","Reymonta
↪  9"),("Tom","Ruska 8")]
True
fromList [("Bob","Lipowa"),("Tim","Reymonta"),("Tom","Ruska")]
fromList [("Tim","Reymonta 9"),("Tom","Ruska 8")]
[("Bob","Lipowa 2"),("Tim","Reymonta 9"),("Tom","Ruska 8")]
```

`fromListWith` is very useful when we have duplicate keys. `fromList` just discards any duplicates. `fromListWith` however allows us to do something different instead.

```
import qualified Data.Map                    as Map

addresses =
  [ ("Bob", "Lipowa 2")
  , ("Tom", "Ruska 8")
  , ("Tim", "Reymonta 9")
  , ("Bob", "Kwiatowa 3")
  , ("Tom", "Nawojki 123")
  ]

m = Map.fromListWith (++) $ map (\(k, v) -> (k, [v])) addresses


main = do
  print $ Map.lookup "Bob" m
  print $ Map.lookup "Tom" m
```

```
Just ["Kwiatowa 3","Lipowa 2"]
Just ["Nawojki 123","Ruska 8"]
```