1) Используя функции ФВП над списками: `map`, `filter`, а также вспомогательные функции, такие как `sum`, `product`, `even` и т.д. напишите выражение, которое:

   a. Вычисляет сумму квадратов элементов списка `[1..10]`

```
sum (map (^2) [1..10])
```

   b. Вычисляет произведение чётных чисел в списке `[4,5,-2,10,11,4,5,8,6]`

```
product (filter even [4, 5, -2, 10, 11, 4, 5, 8, 6])
```

2) Перепишите выражение используя **сечения**:

   a. `map (\x -> x + 5) [1..10]`

```
map (+5) [1..10]
```

   b. `filter (\y -> 5 > y) [3..7]`

```
filter (5>) [3..7]
```

3) Приведите примеры функций, который имеют следующие типы:

   a. `(Float -> Float) -> Float`

```
ghci> :{
ghci| dist' :: (Float, Float) -> Float
ghci| dist' (a, b) = abs(b - a)
ghci| :}
```

   b. `Float -> (Float -> Float)`

```
ghci> :{
ghci| -- compare celsius, kelvin and fahrengeit
ghci| celsius_to_other :: Float -> (Float, Float)
ghci| celsius_to_other arg = (1.8 * arg + 32, arg + 273.15)
ghci| :}
ghci> celsius_to_other 0
(32.0,273.15)
```

   c. `(Float -> Float) -> (Float -> Float)`

```
ghci> :{
ghci| transformation :: (Float, Float) -> (Float, Float)
ghci| transformation (x, y) = (x - 5, y - 5)
ghci| :}
ghci> transformation (1, 2)
(-4.0,-3.0)
```

   d. `a -> (a -> b) -> b`

```
ghci> :{
ghci| application :: a -> (a -> b) -> b
ghci| application x func = func x
ghci| :}
ghci> application 1 (+1)
2
```

e. `a -> b -> c`

f. `(a -> a -> b) -> a -> b`

g. `(a -> b) -> (b -> c) -> (a -> c)`

```
ghci> func f g = f . g
ghci> :t func
func :: (b -> c) -> (a -> b) -> a -> c
```

h. `(a -> b -> c) -> (a -> b) -> (a -> c)`

4) Реализуйте функции:

a. `curry' :: ((a, b) -> c) -> (a -> b -> c)`

```
ghci> curry' f a b = f (a, b)
ghci> :t curry'
curry' :: ((a, b) -> t) -> a -> b -> t
```

b. `uncurry’ :: (a -> b -> c) -> ((a, b) -> c)`

```
ghci> uncurry' f (a, b) = f a b
ghci> :t uncurry'
uncurry' :: (t1 -> t2 -> t3) -> (t1, t2) -> t3
```

5) Используя оператор композиции `(.)` и оператор применения функции `($)` перепишите выражение без использования скобок: `show (sum (map (*3) [1..3]))`.

```
ghci> show . sum $ map (*3) [1..3]
"18"
```

6) Выполните задание:

a. Напишите функцию, которая строит список чисел Фибоначчи.

`fibs : [Integer]`

Не используйте при реализации хвостовую рекурсию!

```
ghci> fibs = 0:1:zipWith(+) fibs (drop 1 fibs)
ghci> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

b. Вычислите список первых 1000 чисел Фибоначчи.

```
take 1000 fibs
```

c. Приведите значение 1000 числа Фибоначчи.

```
ghci> fibs !! 1000
434665576869374564356885276750406258025646605173717804024817290895365554179490518904
038798400792551692959225930803226347752096896232398733224711616429964409065331879382
98969649928516003704476137795166849228875
```

7) Используя свёртки определите следующие функции:

a. concat :: [[a]] -> [a]

```
ghci> concat list_of_lists = foldr (++) [] list_of_lists
ghci> concat [[1, 2], [3, 4], [5, 6]]
[1,2,3,4,5,6]
```

b. inits :: [a] -> [[a]] – список начал списка

```
ghci> inits' list = foldr (\x y -> []:(map(x:) y)) [[]] list
ghci> inits' [1, 2, 3, 4]
[[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

c. tails :: [a] -> [[a]] – список хвостов списка

```
ghci> tails' list = foldr (\x y -> (( x: (head y)): y)) [[]] list
ghci> tails' [1, 2, 3, 4, 5]
[[1,2,3,4,5],[2,3,4,5],[3,4,5],[4,5],[5],[]]
```

8) Треугольные и пирамидальные числа:

a. **Треугольное число** – число монет, которые можно расставить в виде правильного треугольника.

Пример:

$n_1 = 1$: o

$n_2 = 3$: o
o o

$n_3 = 6$: o
o o
o o o

и т.д.

Напишите функцию, которая строит список треугольных чисел:

```
triangulars : [Int]
```

```
ghci> :{
ghci| triangular :: Int -> Int
ghci| triangular x = x * (x + 1) `div` 2
ghci| :}
ghci> triangular 3
6
ghci> :{
ghci| triSeries :: Int -> [Int]
ghci| triSeries x = map triangular [1..x]
ghci| :}
ghci> triSeries 5
[1,3,6,10,15]
```

b. **Пирамидальное число (тэтраэдальное)** – количество шариков, которые можно расположить в виде пирамиды с треугольной равносторонней гранью.

   Напишите функцию, которая строит список пирамидальных чисел:

   `pyramidal : [Int]`

```
ghci> :{
ghci| tetrahedral :: Int -> Int
ghci| tetrahedral x = x * (x + 1) * (x + 2) `div` 6
ghci| :}
ghci> :{
ghci| tetraSeries :: Int -> [Int]
ghci| tetraSeries x = map triangular [1..x]
ghci| :}
ghci> tetraSeries 5
[1,3,6,10,15]
```

9) Напишите рекурсивную функцию, которая подсчитывает число способов разменять сумму с использованием заданного списка номиналов монет.

   Например, есть `3` способа разменять `4`, если у вас есть монеты достоинством `1` и `2`: `1+1+1+1`, `1+1+2`, `2+2`.

   Для выполнения задания реализуйте функцию, которая принимает сумму для размена и список уникальных номиналов монет:

   `countChange :: Int -> [Int] -> Int`,

   а возвращает число способов разменять данную сумму с использованием данных номиналов.

```python
# m in this case is the list of coins
def count( n, m ):
    if n < 0 or len(m) <= 0: #m < 0 for zero indexed programming languages
        return 0
    if n == 0: # needs be checked after n & m, as if n = 0 and m < 0 then it would return 1, which should not be the case.
        return 1
    return count( n, m[:-1] ) + count(n - m[-1], m )
```

```haskell
countChange sum' coins
    | (sum' == 0)    = 1
    | (null coins)   = 0
    | (sum' < 0)     = 0
    | otherwise  = countChange (sum' - (head coins)) coins + countChange sum' (tail coins)
```

Алгебраические типы данных.

Определение: Алгебраический тип данных состоит из суммы произведений типов.

Определения алгебраических типов данных в Haskell имеет следующий вид:

`data TN = TC1 T11 T12 … T1n1 | TC2 T21 T22 … T2n2 | … | TCm Tm1 … Tm(nm),`

`TN` – имя вводимого типа, начинается с прописной буквы.

`TC1 … TCm` – **конструкторы значений типа данных** `TN`, тоже начинаются с прописной буквы.

`Tk1 … Tk(nk)` – `nk` типов аргументов конструктора `TCk` значений типа `TN`.

Замечания:

- конструктор значений типа может вовсе не иметь параметров, т.е. nk (количество аргументов конструктора) может быть равно нулю.
- "|" - надо читать как "или", т.е. значение типа TN – это или TC1 v11 x12 … v1n1, или TC2 v21 v22 … v2n2 или …. TCm vm1 … v(nm), где TCi – это функция, которая принимает (ni) аргументов и создаёт значение типа TN, а vij – это значение типа Tij, где 1 <= i <= m, а 1<= j <= (ni).

Определяемый тип может также иметь типовые переменные, в таком случае определение записывается следующим образом:

`data TN x1 … xn = …,`

где x1 … xn – это типовые переменные, которые могут быть использованы в качестве типов для аргументов конструкторов значений определяемого типа данных.

При этом TN называется **конструктор типа данных**, т.е. функция на типах, которая имеет n аргументов и при передаче фактических типов возвращает новый тип данных

TN v1 …. vn,

где v1…vn – это фактические типы, подставленные вместо переменных типов x1…xn.

10) Выполните задания:

a. Определите алгебраический тип данных `Set a`, который определяет множество элементов типа a.

Напоминание:

Множество – это совокупность объектов, хорошо **различимых** нашей интуицией и мыслимых как единое целое.

Следствие: Любой элемент может входить в множество только один раз!

b. Определите функцию subset :: Eq a => Set a -> Set a -> Bool, которая проверяет, что все элементы первого множества также являются элементами второго множества.

c. Используя функцию subset определите экземпляр класса Eq для типа Set a.

11) Выполните задание:

a. Определите класс типов Finite, который имеет только один метод: получение списка всех элементов заданного типа. Идея в том, чтобы такой список был конечным.

b. Определите экземпляры класса типов Finite для следующих типов:

- Bool
- (a, b) для конечных типов a и b
- Set a (из предыдущего упражнения), где a – конечный тип.
- a -> b, для всяких конечных типов a и b, где класс a также поддерживает равенство. Используя полученное определение создайте также экземпляр класса Eq для типа a -> b.

Класс типов Num.

Класс типов **Num** - самый общий класс числовых типов данных или типов данных, которые представляют собой кольцо. Т.е. значения этих типов данных (числа) можно складывать и умножать.

Определение класса типов Num:

```
class Num a where
  (+)     :: a -> a -> a
  (-)     :: a -> a -> a
  (*)     :: a -> a -> a
  negate :: a -> a
  abs     :: a -> a -- модуль числа
  signum :: a -> a  -- нормирование числа. Для действительных чисел – их
  знак, для нуля – ноль, для комплексных и мультиплексных чисел –
  нормированное число, т.е. x / |x|. Нормирование заключается том, что |x|
  = 1, если x != 0.
```

12) Определите алгебраический тип данных `Complex` для комплексных чисел. Создайте селекторы `realPart` и `imagPart`, которые возвращают действительную и мнимую части комплексного числа соответственно. Complex должен быть экземпляром классов типов Eq и Show.

```
ghci> :{
ghci| data (Complex a) = Complex
ghci|     { realPart :: a,
ghci|       imagPart :: a
ghci|     }
ghci|     deriving (Eq, Show)
ghci| :}
ghci> a = Complex Int 2 3

<interactive>:330:13: error:
    * Illegal term-level use of the type constructor or class `Int'
    * imported from `Prelude' (and originally defined in `GHC.Types')
    * In the first argument of `Complex', namely `Int'
      In the expression: Complex Int 2 3
      In an equation for `a': a = Complex Int 2 3
ghci> a = Complex 2 3
ghci> a
Complex {realPart = 2, imagPart = 3}
```

Определите экземпляр класс типов `Num` для типа `Complex`.

```
data (Complex a) = Complex
    {realPart :: a,
     imagPart :: a
    }
    deriving (Eq, Show)

instance (Num a, Eq a, Floating a) => Num (Complex a) where
    (+) (Complex realPart1 imagPart1) (Complex realPart2 imagPart2) = Complex (realPart1 + realPart2) (imagPart1 + imagPart2)
    (-) (Complex realPart1 imagPart1) (Complex realPart2 imagPart2) = Complex (realPart1 - realPart2) (imagPart1 - imagPart2)
    (*) (Complex realPart1 imagPart1) (Complex realPart2 imagPart2) = Complex (realPart1 * realPart2 - imagPart1*imagPart2)
                                                                              (realPart1 * imagPart2 + imagPart1 * realPart2)
    negate (Complex realPart imagPart) = Complex (negate realPart) (negate imagPart)
    abs (Complex realPart imagPart) = Complex (sqrt (realPart * realPart + imagPart * imagPart)) 0
    signum (Complex realPart imagPart) = if (realPart == 0 && imagPart == 0) then Complex 0 0 else Complex realPart imagPart
    fromInteger n = Complex (fromInteger n) 0
```

Классы типов для функторов и аппликативных функторов и монад:

Класс типов `Functor`:

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

Класс типов `Applicative`:

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

Класс типов Monad:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

13) Дано:

```
module Fun where
  newtype Fun a b = Fun {getFun :: a -> b}
  instance Functor (Fun a) where
    fmap g (Fun k) = Fun (g . k)
  instance Applicative (Fun a) where
    pure k = Fun (const k)
    Fun k <*> Fun j = Fun (\x -> k x (j x))
```

Определить экземпляры классов типов `Functor` и `Applicative` для типа данных `Fun`.

Подсказка: попробовать реализовать сначала следующие экземпляры классов типов:

```
instance Functor ((->) a) where
    …
instance Applicative ((->) a) where
    …
```

Чтобы понять какие реализации должны быть в этом случае у функций `fmap`, `pure` и `(<*>)` можно выписать их типы.

Здесь `(->) a b = a -> b`, т.е. `((->) a) :: * -> *` – это частично применённый конструктор типа `(->) :: * -> * -> *`.

В модуле Prelude, который подключается при загрузке интерпретатора, указанные реализации уже есть, поэтому предлагается проделать это вспомогательное упражнение

без загрузки и проверки результатов в интерпретаторе, а только как наводящее упражнение для решения задачи.

14) Докажите, что любая монада – это также функтор и аппликативный функтор.

Указание: для выполнения задания необходимо, используя функции:

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

Реализовать следующие функции:

```
fmap' :: (a -> b) -> m a -> m b
```

```
fmap' f m = m >>= \x -> return (f x)
```

```
pure' :: a -> m a
```

```
pure' = return
```

```
ap'   :: m (a -> b) -> m a -> m b
```

```
ap' mf mx = mf >>= \f -> mx >>= \x -> return (f x)
```

15) Напишите выражение, которое печатает в консоль “Hello world!”.

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

16) Напишите функцию, которая запрашивает из консоли имя, а затем печатает в консоль:

“Good day, имя”

```
ghci> :{
ghci| main :: IO()
ghci| main = do
ghci|     putStrLn "Name = "
ghci|     name <- getLine
ghci|     let message = "Good day, " ++ name
ghci|     print message
ghci| :}
ghci> main
Name =
Ilya
"Good day, Ilya"
```