1. Используя функции ФВП над списками: map, filter, а также вспомогательные функции, такие как sum, product, even и т.д. напишите выражение, которое:
   1. Вычисляет сумму квадратов элементов списка [1..10]



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



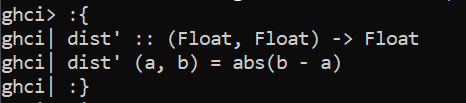
1. Перепишите выражение используя **сечения**:
   1. map (\x -> x + 5) [1..10]



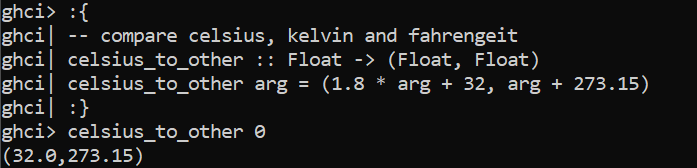
* 1. filter (\y -> 5 > y) [3..7]



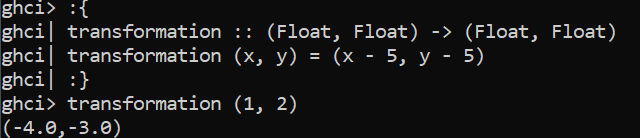
1. Приведите примеры функций, который имеют следующие типы:
   1. (Float -> Float) -> Float



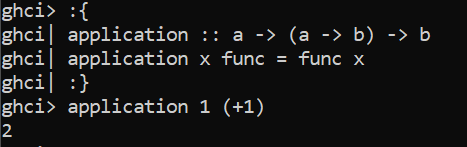
* 1. Float -> (Float -> Float)



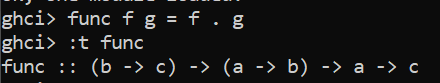
* 1. (Float -> Float) -> (Float -> Float)



* 1. a -> (a -> b) -> b

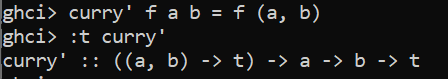


* 1. a -> b -> c
  2. (a -> a -> b) -> a -> b
  3. (a -> b) -> (b -> c) -> (a -> c)

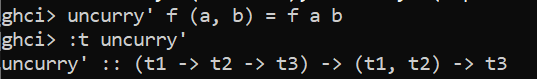


* 1. (a -> b -> c) -> (a -> b) -> (a -> c)

1. Реализуйте функции:
   1. curry' :: ((a, b) -> c) -> (a -> b -> c)



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



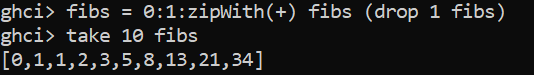
1. Используя оператор композиции (.) и оператор применения функции ($) перепишите выражение без использования скобок: show (sum (map (\*3) [1..3])).



1. Выполните задание:
   1. Напишите функцию, которая строит список чисел Фибоначчи.

fibs : [Integer]

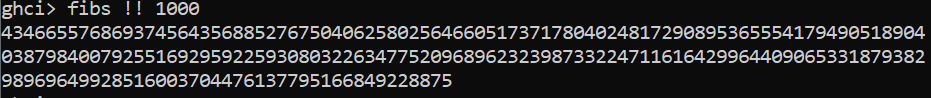
**Не используйте при реализации хвостовую рекурсию!**



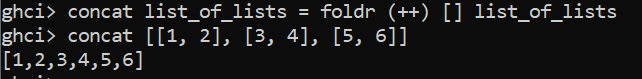
* 1. Вычислите список первых 1000 чисел Фибоначчи.



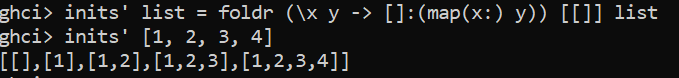
* 1. Приведите значение 1000 числа Фибоначчи.



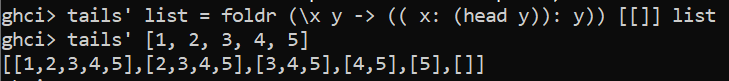
1. Используя свёртки определите следующие функции:
   1. concat :: [[a]] -> [a]



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



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



1. Треугольные и пирамидальные числа:
   1. **Треугольное число** – число монет, которые можно расставить в виде правильного треугольника.

**Пример**:

n­1 = 1: o

n­2 = 3: o

o o

n­3 = 6: o

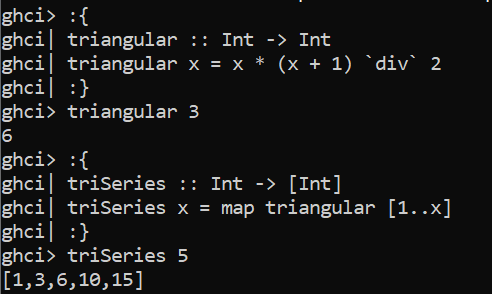
o o

o o o

и т.д.

Напишите функцию, которая строит список треугольных чисел:

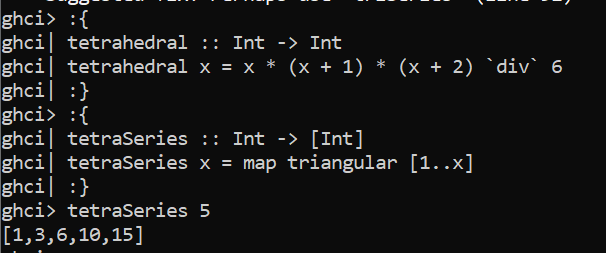
triangulars : [Int]



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

Напишите функцию, которая строит список пирамидальных чисел:

pyramidal : [Int]



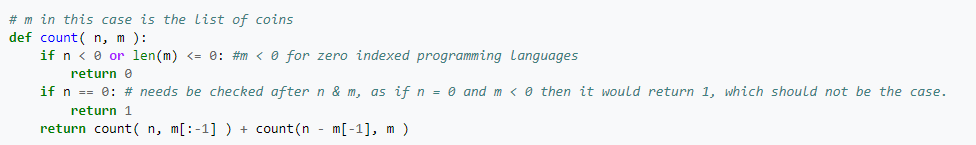
1. Напишите рекурсивную функцию, которая подсчитывает число способов разменять сумму с использованием заданного списка номиналов монет.

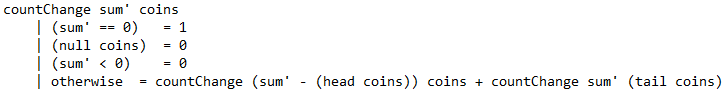
Например, есть 3 способа разменять 4, если у вас есть монеты достоинством 1 и 2: 1+1+1+1, 1+1+2, 2+2.

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.





|  |
| --- |
| **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. |

1. Выполните задания:
   1. 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!

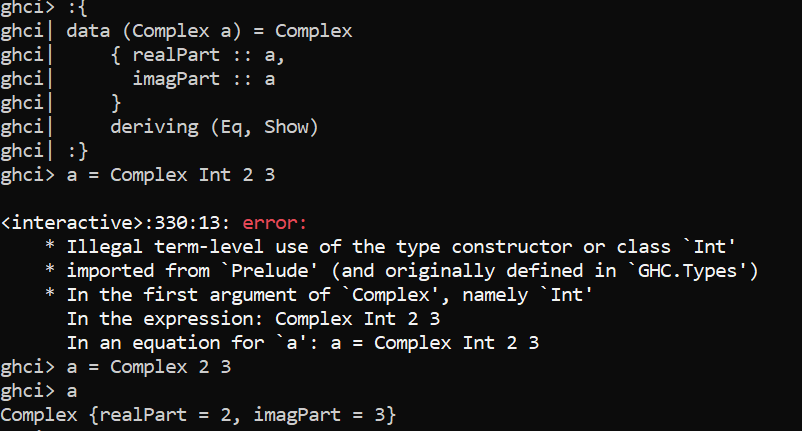
* 1. Определите функцию subset :: Eq a => Set a -> Set a -> Bool, которая проверяет, что все элементы первого множества также являются элементами второго множества.
  2. Используя функцию subset определите экземпляр класса Eq для типа Set a.

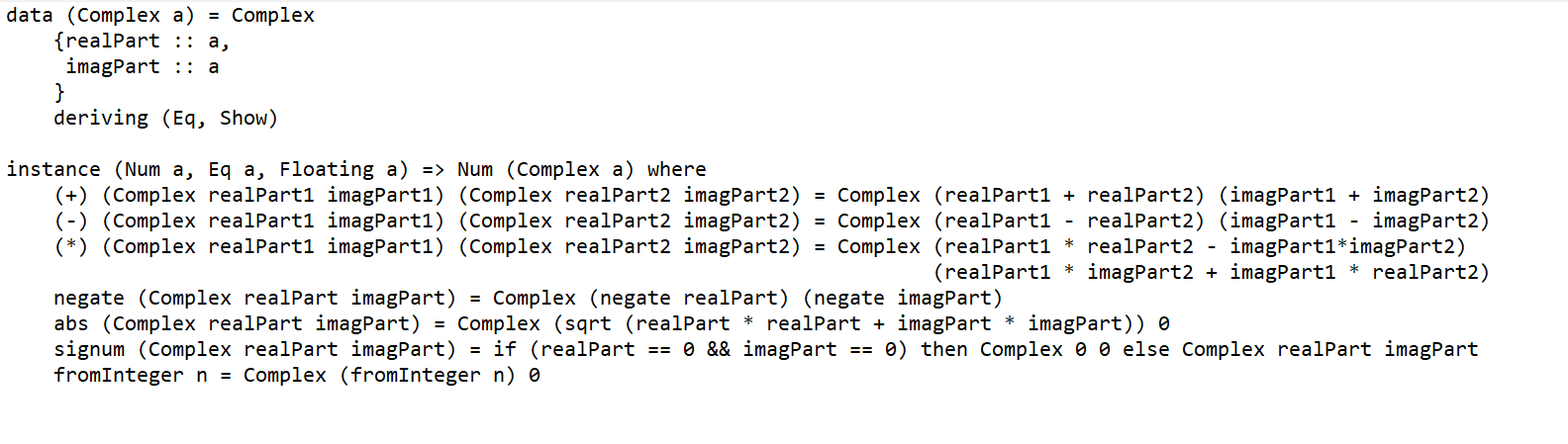
1. Выполните задание:
   1. Определите класс типов Finite, который имеет только один метод: получение списка всех элементов заданного типа. Идея в том, чтобы такой список был конечным.
   2. Определите экземпляры класса типов Finite для следующих типов:

* 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. |

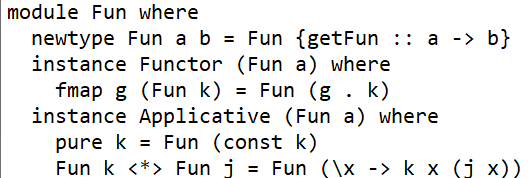
1. Определите алгебраический тип данных Complex для комплексных чисел. Создайте селекторы realPart и imagPart, которые возвращают действительную и мнимую части комплексного числа соответственно. Complex должен быть экземпляром классов типов Eq и Show.

Определите экземпляр класс типов Num для типа Complex.



|  |
| --- |
| **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 |

1. Дано:

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.

1. 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



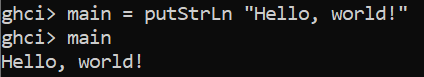
pure' :: a -> m a



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



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



1. Напишите функцию, которая запрашивает из консоли имя, а затем печатает в консоль:

“Good day, имя”

