1. Приведите примеры функций (с реализацией), который имеют следующие типы:
   1. Float −> (Float −> Float)
   2. (Float −> Float) −> (Float −> Float)
   3. a -> (a -> b) -> b
   4. a -> b -> b
   5. (a -> a -> b) -> a -> b
   6. (a -> b) -> (b -> c) -> (a -> c)
   7. (a -> b -> c) -> (a -> b) -> (a -> c)
   8. (a -> b -> b) -> b -> [a] -> b
   9. (b -> a -> b) -> b -> [a] -> b
2. Реализуйте функции:
   1. curry' :: ((a, b) -> c) -> (a -> b -> c)

Функция, которая делает из некаррированной функции каррированную.

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

Функция, которая делает из каррированной функции некаррированную.

1. Используя свёртки определите следующие функции:
   1. concat :: [[a]] -> [a]
   2. inits :: [a] -> [[a]] –- список начал списка
   3. tails :: [a] -> [[a]] –- список хвостов списка
2. 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.

* 1. 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 тип **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. Выполните задание:
   1. Определите алгебраический тип данных **Complex** для комплексных чисел. Создайте селекторы **realPart** и **imagPart**, которые возвращают действительную и мнимую части комплексного числа соответственно. Complexдолжен быть экземпляром классов типов **Eq** и **Show**.
   2. 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. Реализуйте функцию **IncrementAll**, которая получает контейнер-функтор, содержащий числа и увеличивает каждое число в контейнере на 1.

*Примеры*:

incrementAll [1,2,3] ==> [2,3,4]

incrementAll (Just 3.0) ==> Just 4.0

*Шаблон*:

incrementAll :: (Functor f, Num n) => f n -> f n

incrementAll x = undefined

1. Реализуйте функции **fmap2** и **fmap3**, которые выполняют отображение для вложенных функторов.

*Примеры*:

fmap2 on [[Int]]:

fmap2 negate [[1,2],[3]]

==> [[-1,-2],[-3]]

fmap2 on [Maybe String]:

fmap2 head [Just "abcd",Nothing,Just "efgh"]

==> [Just 'a',Nothing,Just 'e']

fmap3 on [[[Int]]]:

fmap3 negate [[[1,2],[3]],[[4],[5,6]]]

==> [[[-1,-2],[-3]],[[-4],[-5,-6]]]

fmap3 on Maybe [Maybe Bool]

fmap3 not (Just [Just False, Nothing])

==> Just [Just True,Nothing]

*Шаблон*:

fmap2 :: (Functor f, Functor g) => (a -> b) -> f (g a) -> f (g b)

fmap2 = undefined

fmap3 :: (Functor f, Functor g, Functor h) =>

(a -> b) -> f (g (h a)) -> f (g (h b))

fmap3 = undefined

1. Напишите функцию, которая складывает два Maybe Int используя операторы **Applicative** (**liftA2** и **pure**). Не используйте сравнение с образцом.

*Примеры*:

sumTwoMaybes (Just 1) (Just 2) ==> Just 3

sumTwoMaybes (Just 1) Nothing ==> Nothing

sumTwoMaybes Nothing Nothing ==> Nothing

*Шаблон*:

sumTwoMaybes :: Maybe Int -> Maybe Int -> Maybe Int

sumTwoMaybes = undefined

1. Напишите функцию, которая получает операцию (**double** или **negate**) и число в виде строк. Функция должна вычислить результат операции. Если передано неизвестное имя функции или некорректное число, то функция должна вернуть Nothing

*Подсказка*: воспользуйтесь функцией **readMaybe** из модуля **Text.Read**.

*Примеры*:

calculator "negate" "3" ==> Just (-3)

calculator "double" "7" ==> Just 14

calculator "doubl" "7" ==> Nothing

calculator "double" "7x" ==> Nothing

*Шаблон*:

calculator :: String -> String -> Maybe Int

calculator = undefined

1. Какое выражение эквивалентно do-блоку:

do y <- z

s y

return (f y)

* 1. z >> \y -> s y >> return (f y)
  2. z >>= \y -> s y >> return (f y)
  3. z >> \y -> s y >>= return (f y)

1. Какой тип у выражения: \x xs -> return (x : xs):
   1. Monad m => a -> [a] -> m [a]
   2. Monad m => a -> [m a] -> [m a]
   3. a -> [a] -> Monad [a]
2. Тип **Result** ниже работает почти как **Maybe**, но в отличие от последнего имеет два типа для обозначения отсутствия значения: один без и один с описанием.

Реализуйте экземпляры классов типов **Functor**, **Monad** и **Applicative** для **Result**.

**Result** ведёт себя как тип **Maybe** в том смысле, что

1. **MkResult** - это как **Just**.
2. Если в процессе вычисления возникает **NoResult**, то всё вычисление даёт в результате **NoResult** (как **Nothing**).
3. Также, если было получено значение **Failure "причина"**, то результат всего вычисления будет **Failure "reason"**.

*Примеры*:

MkResult 1 >> Failure "boom" >> MkResult 2

==> Failure "boom"

MkResult 1 >> NoResult >> Failure "not reached"

==> NoResult

MkResult 1 >>= (\x -> MkResult (x+1))

==> MkResult 2

*Шаблон*:

data Result a = MkResult a | NoResult | Failure String

deriving Show

instance Functor Result where

fmap f result = undefined

instance Applicative Result where

pure = return

(<\*>) = undefined

instance Monad Result where

return = undefined

(>>=) = undefined

1. Ниже дана реализация списков а-ля Haskell. Реализуйте экземпляры классов типов **Functor**, **Applicative** и **Monad** для List.

*Пример*:

fmap (+2) (LNode 0 (LNode 1 (LNode 2 Empty)))

==> LNode 2 (LNode 3 (LNode 4 Empty))

*Шаблон*:

data List a = Empty | LNode a (List a)

deriving Show

instance Functor List where

fmap f result = undefined

instance Applicative List where

pure = return

(<\*>) = undefined

instance Monad List where

return = undefined

(>>=) = undefined

1. Дано:

**module** Fun **where**

**newtype** Fun a b = Fun {getFun :: a -> b}

**instance** Functor (Fun a) **where**

fmap = **???**

**instance** Applicative (Fun a) **where**

pure = **???**

(<\*>) = **???**

Определить экземпляры классов типов **Functor**, **Applicative** и **Monad** 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!”.
2. Напишите функцию, которая запрашивает из консоли имя, а затем печатает в консоль:

“Good day, имя”

1. Реализуйте цикл **while**. **while** должен выполнять операцию до тех пор, пока условие возвращает значение **True**.

*Примеры*:

-- ничего не печатает

while (return False) (putStrLn "IMPOSSIBLE")

-- печатает YAY! до тех пор пока пользователь вводит Y.

while ask (putStrLn "YAY!")

*Шаблоны*:

-- используется в примере

ask :: IO Bool

ask = do putStrLn "Y/N?"

line <- getLine

return $ line == "Y"

while :: IO Bool -> IO () -> IO ()

while cond op = undefined