|  |
| --- |
| **Треугольник Паскаля –** бесконечная таблица биноминальных коэффициентов (Wikipedia), имеющая треугольную форму.  *Пример:*  1  1 1  1 2 1  1 3 3 1  1 4 6 4 1  Он же, но другое выравнивание:  1  **1 1**  **1 2 1**  **1 3 3 1**  **1 4 6 4 1**  Числа, которые располагаются по краям треугольника, равны 1. Число, которое расположено внутри треугольника, рассчитывается как сумма двух чисел, расположенных выше данного числа. |

1. Реализовать функцию **pascal c r** в файле **hw4.hs** для вычисления значений из треугольника паскаля. Здесь **c –** это номер столбца, а **r –** номер строки. Столбцы и строки отсчитываются начиная с нуля.

*Примеры значений функции*:

pascal 0 2 = 1

pascal 1 2 = 2

pascal 2 4 = 6

Функция **printIt n** из файла **hw4.hs** возвращает строковое представление первых **n** строчек треугольника.

Функция **printItIo n** из файла **hw4.hs** печатает первые **n** строчек треугольника в консоль.

*Требование***:** функция **pascal c r** должна быть рекурсивной.

1. Вычислить значения:
2. printIt 5
3. printIt 20
4. printit 100 (попробовать printItIo 100, медитировать…, медитировать.). Поясните результат или его отсутствие.
5. Реализуйте функцию

gcd::Int->Int->Int,

которая находит наибольший общей делитель своих аргументов.

1. Реализуйте функцию

delete :: Char -> String -> String,

которая принимает на вход строку и символ и возвращает строку, в которой удалены все вхождения символа.

*Пример*: **delete ’l’ "Hello world!"** должно возвращать **"Heo word!"**.

1. Реализуйте функцию

substitute :: Char -> Char -> String -> String,

которая заменяет в строке указанный символ на заданный.

*Пример*: **substitute ’e’ ’i’ "eigenvalue"** возвращает **"iiginvalui"**.

1. Реализуйте с использованием хвостовой рекурсии функцию функцию **reverse**, которая обращает список.

*Замечание*: Все необходимые вспомогательные функции должны быть определены при помощи let-выражения.

*Шаблон*:

reverse :: [a] -> [a]

reverse list = ???

1. Не используя какие-либо встроенные функции, реализуйте с использование хвостовой рекурсии функцию:

assoc :: Int -> String -> [(String, Int)] -> Int

Такую, что

assoc def key [(k1,v1), (k2,v2), (k3,v3);...])

ищет первое **i**, такое что **ki** = **key**. Если такое **ki** было найдено, то значение функции – **vi**, иначе значение функции равно **def**.

*Примеры*:

assoc 0 "william" [

("ranjit", 85), ("william",23), ("moose",44)

] ==> 23

assoc 0 "bob" [

("ranjit",85), ("william",23), ("moose",44)

] ==> 0

1. Напишите рекурсивную функцию:

balance :: String → Bool,

проверяющую балансировку скобок в строке. Как вы должны помнить, строки в Haskell — это список символов **[Char]**.

Например, для следующих строк функция должна вернуть **True**:

(if (zero? x) max (/ 1 x))

I told him (that it’s not (yet) done). (But he wasn’t listening)

А для строк ниже, напротив, функция должна вернуть значение False:

:-)

())(

Последний пример демонстрирует: недостаточно проверить только, что строка содержит равное количество открывающих и закрывающих скобок.

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

**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, на ребре которой распложено n шариков.

Напишите функцию, которая строит список пирамидальных чисел:

pyramidals : [Int]

*Примечание*: не используйте прямую формулу для пирамидальных чисел.

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.

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]
   2. Вычисляет произведение чётных чисел в списке [4,5,-2,10,11,4,5,8,6]
2. Перепишите выражение используя **сечения**:
   1. map (\x -> x + 5) [1..10]
   2. filter (\y -> 5 > y) [3..7]

|  |
| --- |
| **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. Если определить тип:

data Potato = Tomato Int Potato (String -> Int),

какой тип будет у **Tomato**?

1. Potato
2. Int -> Potato -> String -> Int -> Potato
3. Int -> Potato -> (String -> Int) -> Potato
4. Tomato
5. Int -> Potato -> (String -> Int) -> Tomato
6. Int -> Potato -> String -> Int -> Tomato
7. Если определить тип:

data ThreeList = ThreeList [a] [b] [c],

какой тип будет у **ThreeList**?

1. [a] -> [b] -> [c] -> ThreeList
2. a -> b -> c -> ThreeList a b c
3. [a] -> [b] -> [c] -> ThreeList [a] [b] [c]
4. [a] -> [b] -> [c] -> Three List a b c
5. Если определить тип:

data TwoList a b = TwoList {aList :: [a], b:List :: [b]}

какой тип будет у функции **bList**?

1. bList – это не функция
2. TwoList a b -> [b]
3. [b] -> TwoList a b
4. [b]
5. Определите тип **RainbowColor**, значения которого – семь цветов радуги. Параметров у конструкторов значений никаких нет.
6. Определите тип **BusTicket**, который может представлять значения вида:
   1. SingleTicket
   2. MonthlyTicket "January"
   3. MonthlyTicket "December"
7. Определите тип данных **Position**, который состоит из двух значений **x** и **y** типа **Int**. Также определите функции ниже для управления значениями типа **Position**.

*Примеры*:

getY (up (up origin)) ==> 2

getX (up (right origin)) ==> 1

*Шаблон*:

data Position = PositionUndefined

--origin – это Position с x и y равными 0

origin :: Position

origin = undefined

-- getX возвращает значение x из Position

getX :: Position -> Int

getX = undefined

-- getY возвращает значение y из Position

getY :: Position -> Int

getY = undefined

-- up увеличивает значение y у Position на 1

up :: Position -> Position

up = undefined

-- right увеличивает значение x у Position на 1

right :: Position -> Position

right = undefined

1. Дано определение типа данных **Nat** – т.н. представление Пеано для натуральных чисел. Суть представления в том, что есть константа **Zero**, которая соответствует нулю и функция для получения следующего числа **PlusOne**. Реализуйте функции **fromNat** и **toNat**, которые конвертируют значения типа **Nat** в **Int** и обратно.

*Примеры*:

fromNat (PlusOne (PlusOne (PlusOne Zero))) ==> 3

toNat 3 ==> Just (PlusOne (PlusOne (PlusOne Zero)))

toNat (-3) ==> Nothing

*Шаблон*:

data Nat = Zero | PlusOne Nat

deriving (Show,Eq)

fromNat :: Nat -> Int

fromNat n = undefined

toNat :: Int -> Maybe Nat

toNat z = undefined

1. Дано определение типа данных **Bin** – обращенная форма бинарного представления целого числа.

Конструктор **O** – представляет бит ноль, а его параметр – это остальная часть бинарного числа.

Конструктор **I** – представляет бит один, его параметр – это также остальная часть бинарного числа.

Конструктор **End** нужен для обозначения конца бинарного числа. Он не принимает параметров.

Для упрощения вычислений биты бинарного числа записываются при помощи типа данных **Bin** в обратном порядке:

1011 ==> I (I (O (I End)))

110 ==> O (I (I End))

10010110 ==> O (I (I (O (I (O (O (I End)))))))

Реализуйте следующие функции: **prettyPrint**, **fromBin**, **toBin**, которые конвертируют **Bin** в человеко-читаемое строковое представление, **Bin** в **Int** и **Int** в **Bin** соответственно.

*Примеры*:

prettyPrint End ==> ""

prettyPrint (O End) ==> "0"

prettyPrint (I End) ==> "1"

prettyPrint (O (O (I (O (I End))))) ==> "10100"

map fromBin

[

O End, I End,

O (I End),

I (I End),

O (O (I End)),

I (O (I End))

] ==> [0, 1, 2, 3, 4, 5]

fromBin (I (I (O (O (I (O (I (O End)))))))) ==> 83

fromBin (I (I (O (O (I (O (I End))))))) ==> 83

map toBin [0..5] ==>

[

O End,

I End,

O (I End),

I (I End),

O (O (I End)),

I (O (I End))

]

toBin 57 ==> I (O (O (I (I (I End)))))

*Шаблон*:

data Bin = End | O Bin | I Bin

deriving (Show, Eq)

-- функция, которая увеличивает бинарное число на один

inc :: Bin -> Bin

inc End = I End

inc (O b) = I b

int (I b) = O (inc b)

prettyPrint :: Bin -> String

prettyPrint = undefined

fromBin :: Int -> Bin

fromBin = undefined

toBin :: Int -> Bin

toBin = undefined