

# Задание 6. Полугруппы, моноиды и аппликативные функторы

## Полугруппы и моноиды

Будем считать, что **полугруппа** это просто множество  $S$ , на котором определена некоторая ассоциативная операция  $\langle \rangle$ .

Операция на множестве  $A$  является ассоциативной, если для любых трёх элементов  $x, y, z \in A$  выполняется следующее равенство:

$$x \langle (y \langle z) = (x \langle y) \langle z$$

Иными словами, не важно, выполним ли мы сначала  $r = y \langle z$ , а потом  $x \langle r$ , или  $r = x \langle y$ , а потом  $r \langle z$ , результат будет одинаковым.

Свойством ассоциативности обладают сложение и умножение чисел, логические И и ИЛИ. Ассоциативной также является операция объединения множеств.

Пусть даны два множества  $A$  и  $B$ . Тогда их объединением называется множество

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Например:

$$(\{1,2,3\} \cup \{3,4,5\}) \cup \{4,5,6\} = \{1,2,3\} \cup (\{3,4,5\} \cup \{4,5,6\}) = \{1,2,3,4,5,6\}.$$

Несложно догадаться, что *множество множеств* (математики говорят “*система множеств*”) является полугруппой относительно операции  $\cup$ .

Если множество  $S$ , связанное с какой-нибудь полугруппой, содержит элемент  $e$  такой, что

- $e \langle x = x \langle e = x$ , где  $x$  — произвольный элемент из  $S$ ,

то такую полугруппу называют **моноидом**, а  $e$  — нейтральным элементом моноида.

Существует особое множество, которое в объединении с любым другим множеством  $A$  образует  $A$ . Оно не содержит никаких элементов, называется *пустым множеством*, и обозначается как  $\emptyset$ :

$$\{1, 2, 3\} \cup \emptyset = \emptyset \cup \{1, 2, 3\} = \{1, 2, 3\}.$$

Взяв во внимание этот факт, можем заявить, что *система множеств*, включающая пустое множество  $\emptyset$ , также является моноидом относительно операции  $\cup$ .

**Полезное следствие из определения моноида:** если мы знаем, что какое-то множество образует моноид, то работая с ним, мы можем не беспокоиться, что порядок выполнения операций как-то нарушит работу программы.

## Задание 1. Удаление повторяющихся элементов в списке

Классическая задача Computer Science:

“Имеется произвольный лист типа `Eq a => [a]`, требуется удалить из него все повторения”.

Очень простое задание, которое можно было бы решить с использованием функции `Data.List.nub`.

Но мы же не ищем легких путей! Будем решать задачу, используя определение моноида...



Как водится, начнём анализ задачи с частного случая. Зададим список целых чисел [1, 2, 1, 2, 3, 4]. От нас требуется получить список [1, 2, 3, 4] (или любой другой, который будет иметь длину 4 и содержать те же самые элементы).

Как сюда можно притянуть моноиды? Давайте рассмотрим следующие математические выражения:

$$\{1\} \cup \{2\} = \{1, 2\}, \{1, 2\} \cup \{1\} = \{1, 2\}, \{1, 2\} \cup \{2\} = \{1, 2\}, \dots$$

Множества по своей природе не содержат повторяющихся элементов. Если бы мы заменили каждый элемент в исходном списке на одноэлементное множество:

$$[1, 2, 1, 2, 3, 4] \rightarrow [\{1\}, \{2\}, \{1\}, \{2\}, \{3\}, \{4\}],$$

и “склеили” множества, используя операцию  $\cup$ :

$$[\{1\}, \{2\}, \{1\}, \{2\}, \{3\}, \{4\}] \rightarrow \{1\} \cup \{2\} \cup \{1\} \cup \{2\} \cup \{3\} \cup \{4\},$$

то получили бы что-то очень похожее на желаемый результат:

$$\{1\} \cup \{2\} \cup \{1\} \cup \{2\} \cup \{3\} \cup \{4\} \rightarrow \{1, 2, 3, 4\}.$$

Правда, мы получили какое-то “множество”, а нам нужен конкретный список, используемый в языке программирования Haskell.

Если мы введем операцию `union :: Eq a => [a] -> [a] -> [a]`, которая включает в один список элементы из другого, не входящие в него, то сможем имитировать множества с помощью обычных списков.

Мы всегда можем создать произвольный список, содержащий повторяющиеся элементы, применить операцию `union` и получить другой список, также содержащий повторения. Чтобы этого не допустить, будем работать с объединением списков очень аккуратно.

Будем утверждать, что результат вычисления `union x y` является множеством только в том случае, если:

- `x` или `y` являются пустым или одноэлементным списком;
- `x` или `y` являются результатом вычисления `union` от других аргументов.

В таком случае тип данных `Eq a => [a]` образует полугруппу относительно операции `union` (аналогично рассмотренному выше объединению множеств).

Более того, `Eq a => [a]` включает пустой список `[]`, и, так как `union x [] = x`, `union [] x = x`, справедливо считать его также моноидом (относительно `union`).

Теперь мы могли бы написать реализации для классов типов `Semigroup` и `Monoid`:

```
instance Eq a => Semigroup [a] where
    (<>) = union

instance Eq a => Monoid [a] where
    empty = []
    mappend = (<>)
```

Но стандартная библиотека Haskell уже определяет списки как полугруппу и моноид относительно операции конкатенации (“склеивания” списков).

Для таких случаев в Haskell принято создавать типы-обертки, реализующие новое поведение для занятых классов типов (за примерами можно обратиться к [списку реализаций \(instances\) моноида на Hackage](#)).

Введём новый тип `Union a (*)`:

```
newtype Union a = Union { getUnion :: [a] } deriving (Eq, Show)
```

Эта строчка задаёт:

- новый тип данных `Union a`, принимающий один типовый параметр `a`;
- конструктор значений этого типа: функцию `Union` принимающую в качестве параметра список типа `[a]`, и возвращающую значение типа `Union a`;
- функцию `getUnion`, позволяющую из значения типа `Union a` вытащить список типа `[a]`;
- шаблон, с помощью которого можно получить заключённое в `Union` значение через сопоставление с образцом:

```
-- |Например, можно было бы написать реализацию getUnion вручную
-- (автогенерация гораздо удобнее, однако)
getUnion :: Union a -> [a]
getUnion (Union xs) = xs
```

Так как мы использовали ключевое слово `newtype` вместо `data`, значения типа `Union a` во время исполнения программы представляют из себя просто список `[a]`.

Этот тип данных будет представлять списки, над которыми можно осуществлять операцию объединения, аналогичную объединению множеств.

Для этого типа мы можем определить новые реализации классов типов `Semigroup` и `Monoid (*)`:

```
instance Eq a => Semigroup (Union a) where
  -- |Ассоциативная операция - объединение списков-множеств.
  -- |Реализация за вами. ( ^ _ ^ )
  (<>) = undefined
```

```
instance (Eq a) => Monoid (Union a) where
  -- |Нейтральный элемент - пустой список.
  -- |Все значения типа [a] должны оборачивать
  -- |в тип Union a с помощью конструктора Union.
  mempty = Union []
  mappend = (<>)
  -- у Monoid еще есть метод mconcat,
  -- но т.к. мы предоставили реализации mempty и mappend,
  -- Haskell использует реализацию по умолчанию.
```

Так, что от нас хотели в начале?.. Ах да, удаление повторяющихся элементов в списке... Так как мы реализовали `Monoid (Union a)`, решение будет тривиальным (\*):

```
-- | преобразует исходный список в новый, не содержащий повторяющихся
-- | элементов.
-- | Алгоритм:
-- | 1. Преобразуем исходный список в список одноэлементных списков-
-- |    множеств,
-- |    совместимых с операцией объединения списков-множеств.
-- | 2. С помощью последовательности ассоциативных операций объединяем
-- |    все списки в один.
-- |    (Это очень просто, _как сложить/перемножить набор чисел, как
-- |    склеить набор строк_)
-- | 3. В результате (2) получим список-множество, вытащим из него
-- |    обычный список.
-- | Он и будет списком без повторений. :)
unique :: Eq a => [a] -> [a]
unique = undefined
  where
    singleton x = [x]
    listOfUnion = fmap (Union . singleton)
```

## Задание 1. Постановка задачи

Требуется:

1. написать реализации для недоопределённых функций (с `undefined` в правой части);
2. протестировать работу функции `unique` на разных входных данных, вручную (через GHCi) и через тестировочную систему.

## Задание 2. Поиск всех простых чисел не больших чем $n$

Простое число — натуральное число, которое делится без остатка только на 1 и на само себя.

Пример:

- число 2 простое (делится только на 1 и на 2);
- число 4 не простое (делится на 1, на 2, на 4).

Все числа, отличные от единицы, и не являющиеся простыми, называют составными. Любое составное число можно представить как произведение его делителей:

- $42 = 2 \times 3 \times 7$ .

Все простые числа, не большие чем  $n$ , можно легко найти с помощью наивного алгоритма перебора делителей.

Суть алгоритма в следующем:

- имеется последовательность  $s$  натуральных чисел от 2 до  $n$  *включительно*;
- имеется таблица `composites` всех составных чисел от 4 до  $n * n$  *включительно*, полученная путём перемножения всех элементов последовательности  $s$  между собой;
- список простых чисел получается из  $s$  путём удаления из последовательности всех составных чисел, встречающихся в ней.

**Extra:** подумайте, можно ли как-то улучшить работу алгоритма?

## Задание 2. Постановка задачи

Требуется написать функцию `primesupton :: Int -> [Int]`, реализующую поиск всех простых чисел, не больших чем  $n$ , с помощью алгоритма перебора делителей. Функция должна принимать число  $n$  и возвращать

список простых чисел, расположенных от меньшего к большему.

Вспомогательный список всех составных чисел **требуется** получить в форме выполнения некоторой бинарной функции над аппликативным функтором списка.

## Подсказка

```
GHCI> (,) <$> [1,2,3] <*> [1,2,3]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```

```
GHCI> (+) <$> [1,2,3] <*> [1,2]
[2,3,3,4,4,5]
```

```
GHCI> import Control.Applicative (liftA2)
GHCI> liftA2 (,) [1,2,3] [1,2,3]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```