monoid.lhs

```
> {-# OPTIONS
>   -Wall
>   -Wno-missing-signatures
>   -Wno-type-defaults
> #-}
```

(ругаемся на всё подряд, но молчим об опущенных сигнатурах и дефолтинге — удобства лишь ради)

Тема

```
> module Monoid where
```

В этом модуле мы рассмотри мы рассмотрим два очень важных для экосистемы Haskell класса типов — `Semigroup` и `Monoid`.

(импорты)

```
> import Data.Monoid
```

Semigroup

Этот класс объединяет типы, для которых существует некая операция — обозначим её `(<>)` — которая позволяет скомпоновать два значения в одно. При этом для каждого типа операция должна обладать свойством ассоциативности:

```
a <> b <> c == (a <> b) <> c == a <> (b <> c)
```

Сам класс выглядит так:

```
class Semigroup a where
  (<>) :: a -> a -> a
  -- я не привожу всех методов класса, если интересно,
  -- смотрите документацию
```

Для списка инстанс Semigroup выглядит так:

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

Т.е. для списка операцией комбинирования выбрана конкатенация. Конкатенация ассоциативна:

```
> listConcatenation = check "List concatenation"
>   [ [1, 2, 3, 4, 5, 6] == [1, 2] ++ ([3, 4] ++ [5, 6])
>   , [1, 2, 3, 4, 5, 6] == ([1, 2] ++ [3, 4]) ++ [5, 6]
>   ]
```

А так выглядит инстанс полугруппы для Maybe:

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> b       = b
    a       <> Nothing = a
    Just a  <> Just b  = Just (a <> b)
```

Заметьте, что от содержимого Maybe требуется реализация Semigroup! Примеры использования:

```
> maybeSemigroup = check "Maybe semigroup"
>   [ Just "foo" <> Just "bar" == Just "foobar"
>   -- содержимое скомбинировалось!
>   , Nothing    <> Just "a"   == Just "a"
>   ]
```

Случается, что для некоторого типа возможны несколько разных ассоциативных операций. В таких случаях инстанс Semigroup для самого типа не делают, а вместо этого определяют несколько типов-обёрток и инстансы для них. Вот пример двух обёрток для числа:

```
data Sum a = Sum a
data Product a = Product a

instance Num a => Semigroup (Sum a) where
  Sum x <> Sum y = Sum (x + y)

instance Num a => Semigroup (Product a) where
  Product x <> Product y = Product (x * y)
```

```
> sumAndProductSemigroup = check "Sum and Product semigroups"
>   [ Sum 40     <> Sum 2     == Sum 42
>   , Product 40 <> Product 2 == Product 80
>   ]
```

Monoid

Этот класс объединяет типы, для которых существует не только ассоциативная операция, но и нейтральный, по отношению к ней, элемент.

Класс типов выглядит так:

```
class Semigroup a => Monoid a
  -- все моноиды — полугруппы
  -- (но не все полугруппы — моноиды!)
  mempty = a
```

Закон нейтральности для моноида выражается так:

```
a <> mempty == mempty <> a == a
```

Для конкатенации списка нейтральный элемент — пустой список, что неудивительно. А вот для `Sum` и `Product` нейтральные элементы различаются! Посмотрим:

```
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  -- x + 0 == 0 + x == x

instance Num a => Monoid (Product a) where
  mempty = Product 1
  -- x * 1 == 1 * x == x
```

Нейтральный элемент бывает полезен, например, когда мы хотим “схлопнуть” список значений, тип которых является моноидом. В этом случае `mempty` выступает начальным значением для свёртки и используется. если вдруг список оказался пуст. А вот для “непустого списка” `NonEmpty` хватает `Semigroup`!

Для схлопывания списков класс `Monoid` имеет отдельный метод:

```
class Monoid a where
  mconcat :: [a] -> a
  mconcat = foldr (<>) mempty
  -- метод уже реализован, но конкретные инстансы могут переопределить
  -- его, если вдруг это будет более эффективно.
```

Композируемость

И `Semigroup` и `Monoid` полезны именно благодаря тому, что они могут композироваться. Скажем, вы можете посчитать сумму и произведение элементов списка за один проход с помощью `Sum`, `Product` и инстанса `Monoid` для пары:

```
> monoidComposition = check "Monoid composition"
>   [ process [1, 2, 3, 4] == (Sum 10, Product 24) ]
>   where
>     process = mconcat . map (\x -> (Sum x, Product x))
```

Задание

Дан тип

```
> data These a b
>   = This a
>   | That b
>   | These a b
```

Реализуйте для него инстансы `Semigroup` и `Monoid` по аналогии с инстансами для `Maybe`. Важно: нужно всё сделать так, чтобы оба “схлопывались”:

```
This "foo" <> That (Sum 40) <> These "!" (Sum 2) == These "foo!" (Sum 42)
```

Ещё важно: проверьте законы! Не факт, что оба инстанса реализуемы :)

Ссылки

- Semigroup
- Monoid

(машинерия для запуска проверок)

```
> check :: String -> [Bool] -> IO ()
> check msg checks
>   | and checks = putStrLn $ "Ok: " <> msg
>   | otherwise  = error $ "Failed: " <> msg

> main = do
>   listConcatenation
>   maybeSemigroup
>   sumAndProductSemigroup
>   monoidComposition
>   putStrLn "ok"
```