Функциональное программирование. Лабораторная работа №1

Задание 1 (2 балла)

1. Напишите функцию `divisors :: Integer -> Integer`, которая принимает в качестве аргумента целое число и возвращает список его делителей (в произвольном порядке).

2. Число называется совершенным, если оно равно сумме своих делителей, включая 1 и исключая само себя. Напишите функцию `isPerfect :: Integer -> Bool`, принимающую целое число и возвращающую `True`, если это число является совершенным, и `False` в противном случае. Используйте функцию `divisors`.

Задание 2 (3 балла)

1. Напишите функцию `splitList :: Int -> [a] -> ([a], [a])`, такую, что вызов `splitList n list` вернет кортеж из двух списков, первый из которых содержит первые `n` элементов списка `list`, а второй — все остальные его элементы.

2. Напишите функцию `splitBy :: (a -> Bool) -> [a] -> ([a], [a])`, которая также разбивает список на две части. Вызов `splitBy condition list` возвращает кортеж из двух списков `(list1, list2)`. Элементы списка `list` включаются в список `list1` до тех пор, пока функция `condition` не вернет для очередного элемента значение `False`; этот элемент и все последующие составляют список `list2`.

3. Напишите функцию `splitWords :: String -> [String]`, разбивающую строку на слова по пробелам, сами пробелы при этом удаляются. Например, вызов `splitWords "abc de cc  e"` должен вернуть список `["abc", "de", "cc", "e"]`. Сравните результаты работы функции `splitWords` с тем, что возвращает стандартная функция `words`.

Полезные функции:

- `take :: Int -> [a] -> [a]` — возвращает указанное число первых элементов списка.
- `drop :: Int -> [a] -> [a]` — отбрасывает указанное число первых элементов списка и возвращает остаток списка.
- `takeWhile :: (a -> Bool) -> [a] -> [a]` — возвращает все элементы из начала списка, которые удовлетворяют условию, до первого не удовлетворяющего условию элемента.
- `dropWhile :: (a -> Bool) -> [a] -> [a]` — отбрасывает все элементы из начала списка, которые удовлетворяют условию.

Задание 3 (5 баллов)

1. Определите тип данных `Expr` для представления арифметических выражений. Каждое выражение может быть вещественной константой (тип `Double`), переменной, а также суммой, разностью, произведением или частным двух выражений.

2. Напишите функцию печати арифметического выражения в инфиксной форме. Не забывайте про скобки (избыточные скобки не являются проблемой).

```
instance Show Expr where
    show expr = ...
```

3. Напишите функцию `eval :: Env -> Expr -> Maybe Double`, вычисляющую значение арифметического выражения.

   Тип данных `Env` описывает контекст, в котором производится вычисление, то есть набор связей между именами переменных и их значениями. Можно использовать любое удобное представление контекста. Одним из наиболее простых представлений является список пар (имя, значение):

```
type Env = [(String, Double)]
```

   Для поиска в такой структуре можно использовать стандартную функцию

```
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
```

   Тип `Maybe a` обычно используется для работы с выражениями, которые могут не иметь значения. Тип `Maybe a` определяется так:

```
data Maybe a = Nothing | Just a
```

   Для работы со значениями типа `Maybe a`, как и с другими алгебраическими типами, можно использовать сопоставление с образцом. Кроме того, может быть полезен оператор `case`:

```
result = case x of
           Nothing -> expression_1
           Just y  -> expression_2
```