parser.lhs

Парсер-комбинаторы

```
> module Parser where

> import Data.Char (isDigit, isSpace)
```

Парсер

```
> newtype Parser a = Parser
>   { runParser
>     :: String
>     -- ^ входная строка
>     -> Maybe (a, String)
>     -- ^ возможный результат разбора и остаток строки
>   }
```

Для парсеров в общем случае хватает реализации Functor + Applicative

```
> instance Functor Parser where
>   fmap f p = Parser $ \s ->
>     case runParser p s of
>       Nothing       -> Nothing
>       Just (x, s')  -> Just (f x, s')

> instance Applicative Parser where
>   pure x = Parser $ \s -> Just (x, s)
>   pf <*> px = Parser $ \s -> do
>     (f, s')  <- runParser pf s
>     (x, s'') <- runParser px s'
>     pure (f x, s'')
>   -- в плане передачи остатка строки из одного подпарсера
>   -- в другой парсер похож на State
```

Примитивы

```
> -- | Возвращает символ, если предикат возвращает True.
> satisfy :: (Char -> Bool) -> Parser Char
> satisfy condition = Parser $ \s ->
>   case s of
>     []     -> Nothing
>     (x:xs) ->
>       if condition x
>       then Just (x, xs)
>       else Nothing

> -- | Ожидает конкретный символ в начале строки (и возвращает его
> -- в случае успеха).
> char :: Char -> Parser Char
> char c = satisfy (== c)

> -- | Ожидает, что строка будет пустой.
> eof :: Parser ()
> eof = Parser $ \s ->
>   case s of
>     "" -> Just ((), "")
>     _  -> Nothing
```

Комбинаторы

```
> -- | Пробудет применить первый парсер, в случае неудачи пробует второй.
> (<|>) :: Parser a -> Parser a -> Parser a
> p1 <|> p2 = Parser $ \s ->
>   case runParser p1 s of
>     r@(Just _) -> r
>     _          -> runParser p2 s

> -- | Возвращает ноль или больше успешных применений парсера
> -- (аналог квантификатора "*" в регулярных выражениях)
> many :: Parser a -> Parser [a]
> many p = many1 p <|> pure []
```

```
> -- | Возвращает одно или больше успешных применений парсера
> -- (аналог квантификатора "+" в регулярных выражениях)
> many1 :: Parser a -> Parser [a]
> many1 p = (:) <$> p <*> many p
```

Примеры использования

С помощью примитивов и комбинаторов уже можно строить парсеры “побольше”.

Начнём с классов символов:

```
> space :: Parser Char
> space = satisfy isSpace

> digit :: Parser Char
> digit = satisfy isDigit
```

Вот наивный парсер целых чисел (не учитывает знак, например):

```
> number :: Parser Int
> number = read <$> many1 digit
```

А это уже более специфичные комбинаторы (те, что выше — общего назначения):

```
> -- | Принимает парсеры-"скобки" и парсер-"содержимое". Возвращает
> -- результат разбора содержимого, при условии что скобки тоже
> -- присутствовали в строке.
> between :: Parser a -> Parser b -> Parser c -> Parser c
> between l r p = l *> p <* r

> -- | Возвращает список (возможно пустой) результатов разбора "значений
> -- разделённых разделителем".
> sepBy :: Parser a -> Parser b -> Parser [b]
> sepBy sep p =
>   ((:) <$> p <*> many (sep *> p))
>   <|>
>   pure []
```

И, наконец, “большой парсер”:

```
> -- | Разбирает строки, содержащие "список в квадратных скобках, содержащий
> -- значения, разделённые запятыми".
> listOf :: Parser p -> Parser [p]
> listOf p =
>   between
>     (spaced $ char '[')
>     (spaced $ char ']')
>     (sepBy (spaced $ char ',') p)
>   where
>     spaced x = many space *> x <* many space
```

Пример использования listOf:

```
> listOfLists :: Maybe [[Int]]
> listOfLists = fmap fst $ runParser (listOf (listOf number) <* eof)
>   "  [ [1, 42,  3  ]  , [42]  ]  "
> -- => Just [[1,42,3],[42]]
```

Задание

Реализуйте инстанс `Monad`.