records.lhs

```
> module Records where
```

Записи, объявление, конструирование

Раньше мы использовали типы-произведения только с позиционными параметрами. В Haskell существует и альтернативный синтаксис для объявления “записей” (records). Выглядит это так:

```
> data Foo a = F
>   --         ^ конструктор значений
>   { bar :: Int
>   --    ^ поле указанного типа
>   , baz :: a
>   --       ^ в типах полей можно использовать переменные типа,
>   -- объявленные в конструкторе типа.
>   }
```

После такого объявления конструктор `F` будет доступен для вызова в виде функции:

```
> foo :: Foo ()
> foo = F 42 ()
```

При этом значения можно будет создавать и в record-синтаксисе:

```
> anotherFoo :: Foo (Foo String)
> anotherFoo = F { bar = 100, baz = F { bar = 42, baz = "lol" } }
```

Кроме конструктора в область видимости добавляются и функции-геттеры

```
bar :: Foo a -> Int
baz :: Foo a -> a

> values :: (Int, Int)
> values =
>   ( bar (F 1 2)  -- 1
>   , baz (F 1 2)  -- 2
>   )
```

Заметьте, имена геттеров должны быть уникальны в пределах модуля, ведь будут сгенерированы одноимённые геттеры. Есть возможности обойти это ограничение, но я пока не буду настолько углубляться в тему.

Вот так иногда решают проблему с уникальностью имён полей:

```
> data User = User
>   { userName :: String
>   , userAge :: Int
>   , userPet :: Pet
>   -- все поля названы с префиксом
>   }

> data Pet = Pet
>   { petName :: String
>   , petKind :: PetKind
>   }

> data PetKind = Dog | Cat
```

Раз геттеры — это функции, их можно композить:

```
> userPetKind = petKind . userPet
```

Обновление записей

Обновление записей означает создание новых значений на основе старых — иммутабельность же! Всегда можно разобрать объект на составляющие с помощью pattern matching, но для records существует свой синтаксис:

```
> bob, agedBob, newBob :: User

> bob = User "Bob" 30 (Pet "Thomas" Dog)

> agedBob = bob { userAge = userAge bob + 1 }
> -- вместо конструктора указано исходное значение, а в скобках указаны
> -- изменяемая поля.
```

```
> newBob = agedBob
>   { userPet = (userPet agedBob) { petName = petName (userPet agedBob) ++ "!" }
>   -- Да, вложенные рекорды обновлять больновато!
>   }
```

Но тут нам могут слегка помочь let-выражения.

let-выражения

До этого мы выносили подвыражения только во `where`-блоки. Но `where` блоки не могут встречаться внутри выражений — только в тех местах, где даются определения. let-выражения же являются именно выражениями, поэтому могут быть использованы, как часть другого выражения!

Вот так выглядит let-выражение:

```
> result :: Int
> result =
>   let
>     -- тут идёт блок определений
>     x = 42
>     sign v
>       | v > 0     = 1
>       | v < 0     = -1
>       | otherwise =  0
>     (y1, y2) = let g v = sign v * v in (g x, g (x - 100))
>     --          ^ однострочный вариант let
>     -- Обратите внимание, что я распаковал пару -- pattern matching
>     -- среди определений тоже возможен!
>   in y1 + y2
> --  ^ in должен быть расположен строго под let, а определения должны быть
> -- сдвинуты "за" let!
```

Так вот, вернёмся к `newBob` и используем let для упрощения кода:

```
> newBob' :: User
> newBob' = agedBob
>   { userPet =
>     let p = userPet agedBob  -- так тоже можно писать, если определение одно
>     in p { petName = petName p ++ "!" }
> --  ^ тут код уже выглядит попроще
>   }
```

Псевдонимы типов

В Haskell существует способ дать сложному типу короткое имя:

```
> type T = Either (Maybe Int) String
```

Только не нужно давать альтернативные имена примитивным типам. Вы не получаете настоящих новых типов, поэтому можно в итоге только запутаться в именах:

```
type FirstName = String
type LastName = String

makePerson :: FirstName -> LastName -> Person
makePerson = …

edison = makePerson ("Edison" :: LastName) ("Thomas" :: FirstName)
-- Никакой дополнительной проверки типов тут не производится,
-- оба псевдонима -- всё ещё просто String.
```

newtypes

Когда нам всё же нужна типобезопасность и хочется различать два вида использований одного и того же типа, используется newtype:

```
> newtype FirstName = FirstName { getFirstName :: String}
> newtype LastName = LastName { getLastName :: String}
> -- по соглашению у таких обёрток геттер называют в стиле getSmth или unSmth
```

У newtype всегда ровно один конструктор и строго одно поле. Но это именно самостоятельный тип. Такие `FirstName` и `LastName` уже не перепутаешь местами — это разные типы!

Но зачем же нужен newtype, если то же самое можно сделать с помощью data?

```
data FirstName = FirstName { getFirstName :: String}
```

Разница в том, что data — это всегда самостоятельное значение со ссылкой на значение поля. И упаковка-распаковка таких обёрток не бесплатна.

А вот newtype после компиляции не существует! Т.е. все заворачивания в `LastName`, паттерн матчинг по этому конструктору, вызовы `getLastName` — всё это не будет в рантайме стоить ничего! Вот поэтому у newtype ровно одно поле — именно оно и останется на месте эфемерной обёртки!

newtypes очень удобны, когда нужно для типа инстанциировать класс, но другой инстанс уже есть, а нам нужно слегка поменять поведение. Или просто напрашиваются несколько инстансов, но какой-то один выбрать нет возможности, ибо инстансы одинаково полезны. В таких случаях делают несколько newtypes и инстанциируют класс для них. Так реализовано большинство типов-моноидов и типов-полугрупп в стандартной библиотеке — да, все эти `Sum` и `Product`!