# Вивчить собі Хаскела на велике щастя!

Автор: Міран Ліповача

Переклад здійснили: Ганна Лелів, Семен Тригубенко, Богдан Пеньковський, Марина Стрельчук і Тетяна Богдан

Мовні редактори: Тетяна Богдан і Ганна Лелів
Науковий редактор: Семен Тригубенко

Переклад виконано за підтримки
Словненя

2017-05-21T00:06:37Z
Версія v4.7-54-gda41cf2

# Зміст

# Розділ 8

# Побудова власних типів і типокласів

*Переклад українською Богдана Пеньковського*

В попередніх розділах ми розглянули деякі вже наявні в мові Хаскел типи і типокласи, а в цьому ми навчимося створювати нові типи і типокласи та залучати їх до роботи!

## 8.1 Вступ до алгебраїчних типів даних

Нам вже траплялося багацько різних типів даних — `Bool`, `Int`, `Char`, `Maybe` тощо. Але як створити новий тип? Що ж, один спосіб — це використати ключове слово **data** для означення типу. Погляньмо, як тип `Bool` означено в стандартній бібліотеці.

```
data Bool = False | True
```

`data` означає, що ми означуємо новий тип даних. Частина, що передує `=` — це ім'я нового типу, і тут ім'я є `Bool`. Речі, які подано після `=`, називаються **конструкторами значень**, і кожен конструктор відповідає значенню, яке цей тип може мати. Знак `|` читається як *або*. Таким чином, ми можемо прочитати це означення отак: тип `Bool` може мати значення `True` або значення `False`. Ім'я типу і конструктори значень повинні починатися із великої літери.

Подібно до означення `Bool`, можна собі уявити, що `Int` означено отак:

```
data Int =
    −2147483648 | −2147483647 | ... | −1 | 0 | 1 | 2 | ... | 2147483647
```

Перший і останній конструктори значень є мінімальним і максимальним значеннями типу `Int`, відповідно. Насправді `Int` означено по-іншому, а три

крапки тут є тому, що ми викинули з означення цілу купу чисел. Це означення — не реальний код, а лише ілюстрація.

А зараз подумаймо про те, яким чином можна **представити** [to represent] геометричну фігуру мовою Хаскел. Наприклад, можна було б використати кортеж. Коло можна представити як `(43.1, 55.0, 10.4)`, де перше і друге числа є координатами центра кола, а третє — радіусом. Наче непогано, але ці числа можна переінтерпретувати як завгодно (наприклад, як тривимірний вектор). Кращий підхід — створити новий тип даних для представлення фігури. Хай фігура може бути або колом, або прямокутником. Маємо:

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Що це таке? Міркуватимемо так. Конструктор значень `Circle` має три **поля** [fields], кожне з яких бере плавомку (`Float`). Отже, означуючи конструктор значень, ми можемо, за бажанням, додати декілька типів після імені того конструктора, і ці типи означуватимуть значення, які «міститиме» цей конструктор. Тут перші два поля є координатами центру, а третє — радіусом. Конструктор значень прямокутника, `Rectangle`, має чотири поля, кожне з яких також приймає плавомку. Перші два є координатами лівого верхнього кута, а два останні — правого нижнього.

Коли я кажу «поля», я насправді маю на увазі параметри. Конструктори значень для якогось типу є звичайними функціями, які повертають значення такого типу. Розгляньмо сиґнатури типів цих двох конструкторів значень.

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Чудово, конструктори значень є функціями, як і все інше. Хто б, скажи, подумав? А тепер запишемо функцію, яка приймає фігуру й повертає площу її

поверхні.

```
surface :: Shape -> Float
surface (Circle _ _ r)          = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Першим гідним уваги тут є оголошення типу. Воно каже: функція отримує фігуру і повертає плавомку. Ми не можемо записати оголошення типу як `Circle -> Float`, тому що `Circle` не є типом — ним є `Shape`. Так само, як ми не можемо написати функцію з оголошеннями типу `True -> Int`. Варто також звернути увагу на таке: ми можемо використовувати конструктори у взірцях. Ми стикалися з цим і раніше (весь час, якщо чесно), коли записували `[]`, або `False`, або `5`, однак ці конструктори не мали жодних полів. Отже, беремо і просто записуємо ім'я конструктора, а потім зв'язуємо його поля з іменами. Оскільки нас цікавить радіус, ми не дуже переймаємося першими двома полями, які нам кажуть, де знаходиться коло.

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Отак — працює! Але якщо ми спробуємо просто надрукувати `Circle 10 20 5` в командному рядку, отримаємо помилку. Це тому, що Хаскел не знає, як серіалізувати в рядок наш новий тип даних (поки що). Пам'ятайте: коли ми намагаємося надрукувати значення в командному рядку, Хаскел спочатку запускає функцію `show`, щоб отримати рядкове представлення цього значення, а вже потім виводить його в терміналі. Щоб зробити наш тип `Shape` членом типокласу `Show`, ми переозначимо його ось як:

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float deriving (Show)
```

Поки що ми не будемо особливо турбуватися про кінцівку цього означення. Просто скажемо що, якщо додати `deriving (Show)` в кінець означення *data*, Хаскел автомагічно зробить цей тип членом типокласу `Show`. І тепер ми можемо виконати таке:

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

Конструктори значень є функціями, тому ми можемо відображати за

їхньою допомогою, частково їх застосовувати, і взагалі робити все що завгодно. Знадобився список концентричних кіл із різними радіусами — легко:

```
ghci> map (Circle 10 20) [4,5,6,7]
[Circle 10.0 20.0 4.0
,Circle 10.0 20.0 5.0
,Circle 10.0 20.0 6.0
,Circle 10.0 20.0 7.0]
```

Наш тип даних хороший, але міг би бути й ще кращим. Запишімо проміжний тип даних, який описує точку у двовимірному просторі. Тоді ми зможемо використати його і зробити наші представлення фігур більш зрозумілими.

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

Зауважимо: означаючи точку, ми назвали тип даних та конструктор значень однаково — `Point`. Тут немає ніякого прихованого змісту, просто, як правило, конструктору надають ім'я таке ж, як і для типу, коли є лише один конструктор значень. Тепер `Circle` має два поля, одне з яких має тип `Point`, а інше — тип `Float`. Це полегшує розуміння в плані «що є що». Те ж стосується і прямокутника. Тепер нам потрібно змінити нашу функцію `surface`, щоб вона могла працювати із новими типами.

```
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) =
    (abs $ x2 - x1) * (abs $ y2 - y1)
```

Єдине, що нам довелося змінити, це — взірці. Ми повністю знехтували точкою у взірці для кола. У взірці для прямокутника ми просто застосували вкладене зіставлення із взірцем, щоб дістати поля точок. Якби нам було треба також поіменувати точки в цілому, можна було б також застосувати взірці з іменами.

```
ghci> surface (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> surface (Circle (Point 0 0) 24)
1809.5574
```

Як щодо функції, яка соватиме фігуру? Вона приймає фігуру, величину, на яку її треба зсунути по осі абсцис, величину зсуву по осі ординат, і повертає нову фігуру із тими ж самими розмірами, от тільки розташовану десь в іншому місці.

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
```

```
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b =
    Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

Досить просто. Для кожного типу фігури ми додаємо зсув до точки, яка описує її місце розташування.

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

Якщо ми не хочемо мати справу безпосередньо з точками, ми можемо написати кілька допоміжних функцій, які спочатку створюватимуть фігури якогось розміру в початку координат, а вже потім зсуватимуть їх куди треба.

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Звісно, у модулях ви можете експортувати не тільки функції, а й типи даних. Для цього достатньо записати ваш тип разом із іменами функцій, які ви експортуєте, а потім додати дужки. В дужках треба вказати розділені комами конструктори значень, які ви хочете експортувати. Якщо ви хочете експортувати всі конструктори значень якогось типу, просто пишіть в дужках дві крапки (себто, `..`).

Якщо б ми хотіли експортувати функції і типи, які ми означили в даному розділі, ми могли б почати з цього:

```
module Shapes
( Point(..)
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where
```

За допомогою `Shape(..)` ми експортували всі конструктори значень для типу `Shape`. Отже, кожен, хто імпортує наш модуль, може будувати фігури, використовуючи конструктори значень `Rectangle` та `Circle`. Це те ж саме, що і `Shape (Rectangle, Circle)`.

Також ми могли б і не експортувати жодних конструкторів значень для `Shape` взагалі, просто записавши `Shape` в інструкції експорту. У такий спосіб, той, хто імпортує наш модуль, зможе створювати фігури тільки за допомогою допоміжних функцій `baseCircle` та `baseRect`. Модуль `Data.Map` використовує цей підхід. Ви не можете створити мапу, виконавши `Map.Map [(1,2),(3,4)]`, тому що відповідний конструктор значень не експортований. Проте ви можете скористатись однією із допоміжних функцій на кшталт `Map.fromList`. Пам'ятаймо, конструктори значень є просто функціями, які отримують поля як параметри й повертають значення відповідного типу (як-от `Shape`) як результат. Тому, коли ми вирішуємо їх не експортувати, ми запобігаємо використанню цих функцій тими, хто імпортує наш модуль. Якщо конструктори значень не експортуються, але експортуються якісь інші функції, які повертають значення потрібного типу, ми можемо використати ці функції для побудови значень.

Типи даних, для яких конструктори не експортовано, є більш абстрактними, бо їхню реалізацію приховано. До того ж, використання конструкторів значень у зіставленнях із взірцем стає неможливим.

## 8.2 Синтаксис для записів

Нехай нам дали завдання створити тип даних, який описуватиме особу. Інформація, яку б ми хотіли зберігати, отака: ім'я, прізвище, вік, зріст, номер телефону і улюблений вид морозива. Не знаю як вам, але це все, що мені треба знати про людину. Поїхали!

```
data Person = Person String String Int Float String String deriving (Show)
```

Гаразд. Першим полем є ім'я, наступним — прізвище, третім йде вік і так далі. Для прикладу, створімо особу:

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

Непогано, але читається нелегко. Що, якби ми хотіли записати функцію, яка б повертала лише якусь окрему частину інформації про особу? Функцію, яка повертає ім'я, функцію, яка повертає прізвище, і так далі. Що ж, тоді б нам довелося означити їх приблизно так:

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

Фух. Написав, але без жодного задоволення! Втім, незважаючи на незграбність цього розв'язку і НУДЬГУ, яку відчуваєш, коли його записуєш, він працює.

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

Ви скажете, що має бути кращий спосіб. Ні — його не існує, вибачте.

Жартую, є. Ха-ха-ха! Творці Хаскела були дуже розумними й передбачили такий хід подій. Вони надали альтернативний спосіб означування таких типів даних. Ось як ми могли б отримати еквівалентний **функціонал** [functionality] за допомогою записів.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

Тому замість простого оголошення типів полів, одного за одним, розділених пробілами, ми користуємося фігурними дужками, а усередині, для кожного поля, спочатку пишемо ім'я поля, наприклад, `firstName`, потім ставимо подвійну двокрапку `::` (вона також зветься Paamayim Nekudotayim[†], хе-хе), а потім вказуємо тип. В результаті маємо тип даних, повністю аналогічний попередньому. Головною перевагою цього підходу є те, що функції для отримання значень полів створюються автоматично. Через те, що ми використали запис, Хаскел нам автоматично створив отакі функції: `firstName`, `lastName`, `age`, `height`, `phoneNumber` та `flavor`.

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

Існує ще одна перевага у використанні записів. Коли **ми автовтілюємо** `Show` [we derive `Show` instance], тип серіалізуватиметься в рядок по-іншому, якщо ми означимо його за допомогою синтаксису для записів. Хай в нас є тип, який **представляє** [represents] автомобіль. Ми хочемо бути в курсі, яка компанія його виготовила, яка назва моделі та який рік виробництва. Дивіться:

```
data Car = Car String String Int deriving (Show)
```

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

Якщо ми переозначимо цей тип, використовуючи запис, ми можемо побудувати авто ось так:

```
data Car = Car {company :: String, model :: String, year :: Int} deriving (Show)
```

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

Створюючи нове авто, не обов'язково подавати поля в правильному порядку — головне, щоб вони всі були подані. Але якщо ми не використовуємо синтаксис для записів в означенні типу, при створенні значення цього типу значення-поля мають бути подані в тому порядку, в якому вони подані в означенні цього типу.

---

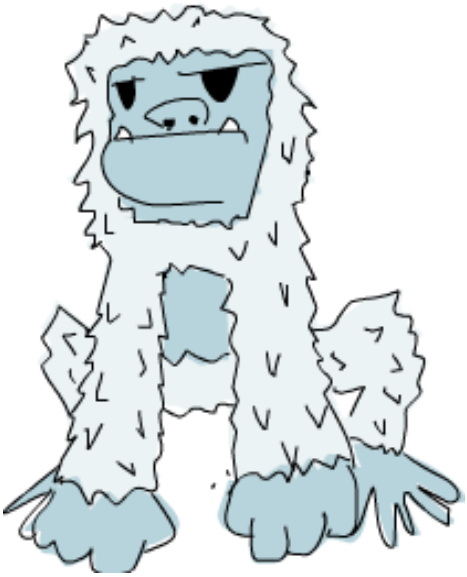
[†] З івриту: נקודתיים פעמיים, вимовляється як [paʔaˈmajim nəkudoˈtajim], і означає «подвійна двокрапка». Ця назва ніде не використовується, окрім як в Zend Engine 0.5 з PHP 3, який розроблено в Ізраїлі і де можна зустріти наповнені змістом повідомлення про помилку на кшталт «Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM» :-). В PHP подвійна двокрапка є оператором визначення зони видимості.

Використовуйте записи, коли конструктор має кілька полів і не є очевидним «що є що». Якщо ми означимо тип даних для тривимірних векторів як `data Vector3D = Vector Int Int Int`, з цього означення легко здогадатися, що поля швидше за все є елементами вектора. Проте в наших типах `Person` та `Car` семантичне навантаження полів сяє не настільки яскраво, і ми значно виграємо від використання синтаксису для записів в цих означеннях.

## 8.3 Параметри типів

Конструктор значень може брати декілька значень-параметрів і повертає нове значення. Наприклад, конструктор `Car` приймає три значення і будує значення типу `Car`. За аналогією, **конструктори типів** також можуть приймати інші типи як параметри і будувати нові типи. Це може спочатку звучати занадто «мета», але насправді це не складно. Якщо вам знайомі **шаблони C++** [templates in C++], то ви помітите, що вони схожі на конструктори типів. Для того, щоб розібратися як працюють параметри типів, розгляньмо реалізацію одного з типів, який нам вже зустрічався:

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



Тут `a` є параметром типу. І через те, що параметр типу є, ми називаємо `Maybe` конструктором типу. Залежно від того, *що* ми хочемо щоб цей тип містив (коли він не є `Nothing`), цей конструктор типів може побудувати нам типи `Maybe Int`, `Maybe Car`, `Maybe String` і так далі. Значення не може мати тип просто `Maybe`, бо це не є тип — це є конструктор типів. Для того, щоб це був справжній тип (себто, можна створити значення, що матиме такий тип), всі його параметри повинні бути заповнені.

Таким чином, якщо ми передаємо `Char` як параметр до `Maybe`, ми отримуємо тип `Maybe Char`. До прикладу, значення `Just 'a'` має тип `Maybe Char`.

Ви могли цього й не знати, але ми вже користувалися типом, що має параметр, ще до того як ми познайомилися із `Maybe`. Цим типом є список! Хоча в грі задіяно певний синтаксичний цукор, тип-список приймає параметр аби утворити конкретний тип. Значення можуть мати тип `[Int]`, або `[Char]`, або `[[String]]`, але не може бути значення, яке має просто тип `[]`.

Повозімося в пісочниці із типом `Maybe`:

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: Num t => Maybe t
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

Параметри типів є корисними, бо вони уможливлюють створення різних типів залежно від того, *що* ми бажаємо розмістити «усередині» тих типів. Коли ми пишемо `:t Just "Haha"`, система виведення типів з'ясовує, що це має бути тип `Maybe [Char]`, адже `a` в `Just a` є рядком, тому `a` в `Maybe a` також має бути рядком.

Зауважте, що `Nothing` має тип `Maybe a`, і цей тип є поліморфним. Якщо деяка функція бере `Maybe Int` як параметр, ми можемо передати їй `Nothing`, бо `Nothing` не містить значення взагалі, тому не важливо, який у цього «відсутнього» значення тип. Тип `Maybe a` може поводитись як `Maybe Int` якщо треба, так само як `5` може поводитись як `Int` або як `Double`. Аналогічно, порожній список має тип `[a]` і може поводитись як порожній список чого завгодно. Саме тому ми можемо обчислювати `[1,2,3] ++ []` та `["ha","ha","ha"] ++ []`.

Параметризація типів — річ корисна, але тільки тоді, коли вона доречна. Зазвичай вона використовується, коли наш тип даних працюватиме незалежно від типу значення, яке він тримає в собі — подібно до `Maybe a`. Якщо наш тип поводиться як «коробка для чогось», то варто цей тип параметризувати. Ми могли б змінити наш тип даних `Car` із цього:

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

на цей:

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
```

Але чи виграли б ми насправді? Відповідь: ймовірно, що ні, бо ми тільки-но означили функції, які працюють лише з типом `Car String String Int`. Наприклад, маючи наше перше означення для `Car`, ми могли б записати функцію, яка виводить на екран характеристики автомобіля гарненьким текстиком.

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

Симпатична невеличка функція! Оголошення типу — симпатичне, функція добре працює. А що, коли `Car` був би `Car a b c`?

```
tellCar :: Show a => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

Ми змусили цю функцію приймати `Car` ось такого типу: `Show a => Car String String a`. Як бачите, сиґнатура типу є складнішою ніж попередня, а єдина користь, яку ми отримуємо у винагороду — це те, що тепер `c` може бути будь-яким типом, який втілює типоклас `Show`.

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: Num t => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

Але в реальності ми швидше за все користуватимемося `Car String String Int` у більшості випадків, тому схоже, що параметризувати тип `Car` не варто. Зазвичай ми використовуємо параметри типів, коли «внутрішні» типи, що їх містять у собі різні конструктори значень «зовнішнього» типу, не є важливими для роботи того «зовнішнього» типу. Список «чогось» є списком чогось і не має значення, яким є тип того чогось — список все одно може працювати. Якщо ми хочемо обчислити суму списку чисел, ми можемо, згодом, у функції сумування, вказати, що ми хочемо саме список *чисел*. Те ж стосується `Maybe`. Тип `Maybe` описує ситуацію, коли є можливість або не мати нічого (наприклад: немає результату обрахунку), або мати щось одне якогось типу (наприклад: є одна-єдина відповідь). Для роботи `Maybe` не має значення, яким є тип того «чогось».

Іншим прикладом параметризованого типу, який нам вже траплявся, є `Map k v` із модуля `Data.Map`. `k` є типом ключів у мапі і `v` є типом значень. Це є хороший приклад структури даних, де параметри типів є дуже корисними. Параметризовані мапи дозволяють нам описувати відображення з будь-якого типу в будь-який інший, за умови, що тип ключа належить до типокласу `Ord`. Якби ми взялися написати означення мапи, то в нас могло б виникнути бажання включити в означення даних цю умову типокласу, на кшталт:

```
data Ord k => Map k v = ...
```

Проте ми маємо дуже серйозну домовленість в Хаскелі: **ніколи не вказувати умови типокласів в означеннях даних**. Чому? А тому, що ми не отримуємо особливої користі від цього, а роботи це додає — змушує писати більше класових обмежень, навіть коли вони не потрібні. Є умова типокласу `Ord k` в *data*-означенні для `Map k v` чи немає — все одно доведеться писати цю умову в сиґнатурах типу функцій, яким потрібно, щоб ключі у мапі можна було впорядкувати. Зате, якщо ми не вкажемо цю умову в означенні, нам не потрібно буде писати `Ord k =>` в оголошеннях типу тих функцій, для яких можливість впорядкування ключів не є необхідністю. Прикладом такої функції є `toList`, яка лише бере мапу і перетворює її в асоціативний список. Її сиґнатурою типу є `toList :: Map k a -> [(k, a)]`. Якщо б `Map k v` мало б умову типокласу в *data*-означенні, тип `toList` мав би бути `toList :: Ord k => Map k a -> [(k, a)]`, хоча ця функція не робить порівнянь ключів взагалі.

Тому не варто записувати умови в *data*-означеннях, навіть коли здається, що в цьому є сенс, оскільки вам все одно доведеться їх записувати в оголоше-

ннях типу функцій.

Реалізуймо тривимірний вектор і деякі операції, що працюють із ним. Скористаймося параметризованим типом: хоча вектор зазвичай містить значення лише чисельних типів, чисельних типів є декілька, і тому параметризація доречна.

```
data Vector a = Vector a a a deriving (Show)

vplus :: Num t => Vector t -> Vector t -> Vector t
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: Num t => Vector t -> t -> Vector t
(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: Num t => Vector t -> Vector t -> t
(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n
```

`vplus` додає два вектори і це реалізовано простим додаванням відповідних компонент. Функція `scalarMult` — для обрахунку скалярного добутку двох векторів, а `vectMult` — для множення вектора на скаляр. Ці функції можуть оперувати `Vector Int`, `Vector Integer`, `Vector Float` і так далі, допоки виконується умова, що `a` із `Vector a` належить типокласу `Num`. Також, перевіривши оголошення типу для цих функцій, ви побачите, що вони можуть оперувати тільки векторами одного типу, а інші додаткові числа, що є залучені в деяких операціях, повинні бути того ж типу, що і числа, які містяться у векторах. Зверніть увагу — ми не додавали умову `Num` в *data*-означення, бо її нам все одно додавати в оголошення типу функцій.

Знову ж таки, дуже важливо розрізняти конструктори типів і конструктори значень. В означенні типу даних, все, що стоїть перед `=`, є конструктором типів, а конструктори значень йдуть після (відокремлені знаком `|`, якщо їх декілька). Тому, наприклад, `Vector t t t -> Vector t t t -> t` — то є нісенітниця через те, що в означенні типу мають бути задіяні типи, а конструктор **типу** вектора приймає лише один параметр, тоді як конструктор **значень** приймає три. Нумо досліджувати наші вектори!

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vectMult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `scalarMult` Vector 9.0 2.0 4.0
```

```
74.0
ghci> Vector 2 9 3 `vectMult` (Vector 4 9 5 `scalarMult` Vector 9 2 4)
Vector 148 666 222
```

## 8.4 Автоматичні втілення

У підрозділі **??** ми пояснили основні принципи дії типокласів. Ми дізналися, що типоклас є схожим на інтерфейс, який означує деяку поведінку. Тип може **втілити** типоклас, якщо він підтримує таку поведінку. Приклад: тип `Int` є втіленням типокласу `Eq`, тому що типоклас `Eq` означує поведінку для речей, які можна порівнювати. А оскільки цілі числа можна порівнювати, `Int` є членом типокласу `Eq`. Справжня користь приходить разом із функціями, які працюють як інтерфейс для `Eq`, а саме — `==` та `/=`. Якщо тип є членом типокласу `Eq`, ми можемо **перевіряти на рівність** [check for equality] значення цього типу за допомогою функції `==`. Ось чому вирази `4 == 4` та `"foo" /= "bar"` **успішно проходять перевірку системою типів** [typecheck] (іншими словами — не містять помилок типу).

Ми також згадували, що типокласи часто плутають із класами з імперативних мов, таких як Java, Python, C++ і таке інше. Це збиває з пантелику багатьох людей. В щойно перелічених мовах класи є «кресленнями», згідно яких конструюються об'єкти, які інкапсулюють в собі стан та можуть виконувати певні дії. Типокласи ж більш за все схожі на інтерфейси. Ми не будуємо об'єкти з типокласів. Натомість ми спочатку створюємо наш тип даних, а вже потім ми думаємо про те, як він може поводитись: якщо як щось, що можна порівнювати, ми робимо його втіленням типокласу `Eq`; якщо він може поводитись як щось, що може бути впорядковане, ми робимо його втіленням типокласу `Ord`; і так далі.

В наступному підрозділі ми розглянемо, як ми можемо вручну робити власноруч створені типи втіленнями типокласів — втілювати типоклас, як ми згодом побачимо, це є те саме, що й означувати функції, означування яких вимагає цей типоклас. Але зараз гляньмо, як Хаскел може втілити отакі типокласи для нас автоматично: `Eq`, `Ord`, `Enum`, `Bounded`, `Show` і `Read`. Наказати Хаскелові автомагічно втілити певну поведінку у наших типах можна за допомогою ключового слова *deriving*, і це ключове слово треба вжити в означенні нашого типу даних.

Розгляньмо цей тип даних:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Він описує особу. Будемо вважати, що немає двох людей із однаковим ім'ям, прізвищем та віком. Тепер, наприклад, хай в нас є два записи для двох людей. Чи має сенс перевірка чи відповідають ці записи одній і тій самій особі? Звісно, що так. Ми можемо спробувати порівняти їх і подивитися, чи однакові вони, чи ні. Ось чому має сенс зробити цей тип членом типокласу `Eq`. На цей раз втілимо `Eq` в `Person` автоматично за допомогою *deriving*:

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

Якщо ми означимо `Person` в такий спосіб, а потім спробуємо порівняти два значення типу `Person` за допомогою `==` чи `/=`, Хаскел спочатку перевірить, чи збігаються конструктори значень (хоча в цьому випадку є тільки один конструктор значень), а потім перевірить, чи однакові всі поля. Тестування кожної пари полів також відбувається за допомогою `==`. Але є одне але: щоб це спрацювало, типи всіх полів також повинні бути членами типокласу `Eq`. Оскільки і `String`, і `Int` вже втілюють `Eq`, проблем не виникає. Тестуймо наше новеньке втілення:

```
ghci> let mikeD = Person { firstName = "Michael"
                         , lastName = "Diamond"
                         , age = 43 }
ghci> let adRock = Person { firstName = "Adam"
                          , lastName = "Horovitz"
                          , age = 41 }
ghci> let mca = Person { firstName = "Adam", lastName = "Yauch", age = 44 }
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person { firstName = "Michael"
                      , lastName = "Diamond"
                      , age = 43 }
True
```

Звісно, оскільки тепер `Person` — в `Eq`, ми можемо використати його замість `a` в усіх функціях, які мають умову типокласу `Eq a` в своїй сиґнатурі типу, як-от, наприклад, `elem`.

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

Типокласи `Show` та `Read` — для речей, які можуть бути перетворені на рядки або відновлені з рядків відповідно. Як і з `Eq`, якщо ми хочемо, щоб якийсь тип втілював `Show` чи `Read`, а конструктор того типу має поля, тоді типи полів також мають бути членами `Show` чи `Read`. Зробімо наш тип даних `Person` на додачу ще й членом типокласів `Show` і `Read`.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

Тепер можна видрукувати особу в терміналі.

```
ghci> let mikeD = Person { firstName = "Michael"
                         , lastName = "Diamond"
                         , age = 43 }
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43
}"
```

Якщо б ми спробували надрукувати особу в терміналі до того, як тип даних `Person` отримав членство в `Show`, Хаскел би поскаржився на нас, заявляючи, що йому не відомо, як можна представити особу рядком. Але оскільки ми автоматично втілили `Show`, йому це тепер відомо.

`Read` є по суті оберненим типокласом до `Show`. Типоклас `Show` існує для перетворення значень на рядки, а `Read` — для перетворення рядків на значення. Проте пам'ятайте: коли ми використовуємо функцію `read`, ми повинні явно анотувати тип, щоб повідомити Хаскелу, значення якого типу ми хочемо мати в результаті. Якщо ми цього не зробимо в явному вигляді, Хаскел не знатиме про який тип йдеться.

```
ghci> let s =
```

```
    "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}"
ghci> read s :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

Якщо ми використаємо результат `read` так, що Хаскел зможе **вивести** [infer], що він має перетворити рядок на `Person`, тоді не потрібно буде анотувати тип явно.

```
ghci> let s =
    "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}"
ghci> read s == mikeD
True
```

Ми також можемо читати параметризовані типи, але тоді необхідно заповнити параметри типу. Тому ми не можемо записати `read "Just 't'" :: Maybe a`, але можемо записати `read "Just 't'" :: Maybe Char`.

Також можна автоматично втілити й `Ord`. Цей типоклас — для типів, значення яких можна впорядкувати. А якщо порівняти два значення одного типу, але такі, що були створені за допомогою різних конструкторів? У цьому випадку вважається меншим значення, яке було побудовано за допомогою конструктора, який було означено першим. Наприклад, розгляньмо тип `Bool`, який може мати значення `False` або `True`. З метою ілюстрації його поведінки при порівнянні значень будемо вважати, що цей тип є реалізований отак:

```
data Bool = False | True deriving (Ord)
```

Оскільки конструктор значень `False` подано першим, а `True` вказано після нього, `True` вважатиметься більшим за `False`.

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True < False
False
```

У типі даних `Maybe a` конструктор значень `Nothing` вказаний перед конструктором значень `Just`, тому значення `Nothing` є завше меншим за значення `Just <<something>>`, навіть коли це `<<something>>` є мінус один мільярд трильйонів. Але якщо ми порівняємо два значення `Just`, то порівнюватимуться речі усередині цих двох `Just`:

```
ghci> Nothing < Just 100
```

```
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

Але от `Just (*3) > Just (*2)` не спрацює, бо `(*3)` та `(*2)` є функціями, а функції не втілюють `Ord`.

Ми можемо запросто використати алгебраїчні типи даних[†] для побудови перелічень — типокласи `Enum` та `Bounded` допоможуть нам в цьому. Розгляньмо оцей тип даних:

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday
```

Оскільки тут всі конструктори значень є **нульарними** [nullary] (не приймають параметрів, тобто полів), ми можемо зробити цей тип даних членом типокласу `Enum`. Типоклас `Enum` об'єднує разом всі речі, в яких є **попередники** [predecessors] та **наступники** [successors]. Ми можемо також зробити його членом типокласу `Bounded`, який об'єднує усі речі, які мають мінімум і максимум. І оскільки ми маємо таку хорошу нагоду, зробімо так, щоб наш тип на додачу втілював усі типокласи, які можна втілити автоматично, і погляньмо, що тепер з цим типом можна робити.

```
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
```

[†]Алгебраїчні типи даних (АТД) називаються алгебраїчними, бо будувати їх дозволено операціями, які разом утворюють таку собі невеличку «алгебру». Алгебра АТД в Хаскелі складається з «сум» (наприклад, в `data Pair = I Int | B Bool`, маємо розгалуження — структура даних `Pair` є або `Int` із конструктором `I` або `Bool` із конструктором `B`) і «добутків» (наприклад, в `data Pair = P Int Bool`, маємо поєднання — структура даних `Pair` є поєднанням `Int` і `Bool` за допомогою конструктора `P`).

```
        | Sunday
          deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

Завдяки членству в `Show` та `Read`, ми можемо перетворити значення цього типу на рядки — і навпаки.

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

Дякуючи типокласам `Eq` та `Ord`, ми можемо порівнювати та впорядковувати дні.

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

Через те, що тип є членом `Bounded`, ми можемо знайти найменший день і день найбільший.

```
ghci> minBound :: Day
Monday
ghci> maxBound :: Day
Sunday
```

Не забуваймо про `Enum`. Ми можемо знаходити попередників та наступників днів та використовувати дні в побудові діапазонів днів!

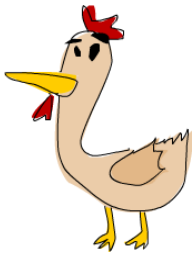
```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

Круто!

## 8.5 Типи-синоніми

Ми вже згадували, що `[Char]` та `String` є еквівалентними та взаємозамінними. Це реалізовано за допомогою **синонімії типів**. Типи-синоніми по суті нічого не роблять: синонімія існує для того, щоб можна було давати типам різні імена, для поліпшення читабельності коду і документації. Ось як стандартна бібліотека означує тип-синонім `String`, який є синонімом типу `[Char]`.

```
type String = [Char]
```



Ми ввели нове ключове слово *type*. Воно може збити декого із пантелику, бо ми взагалі-то не створюємо нічого нового (ми це робили за допомогою ключового слова *data*). Натомість ми просто створюємо синонім для типу, який вже існує.

Якщо ми придумаємо функцію, яка переганяє рядок у верхній регістр і назвемо її `toUpperString`, або ще щось, то ми можемо оголосити її як `toUpperString :: [Char] -> [Char]` або ж як `toUpperString :: String -> String`. Обидва варіанти є по суті ідентичними, але останній приємніше читати.

Коли ми мали справу із модулем `Data.Map`, ми спочатку представили телефонну книгу асоціативним списком, а потім поміняли представлення на мапу. Як ми уже з'ясували, асоціативний список — то є список пар ключ-значення. Тож погляньмо на телефонну книгу, яка у нас була.

```
phoneBook :: [(String,String)]
phoneBook =
    [("betty","555-2938")
    ,("bonnie","452-2928")
    ,("patsy","493-2928")
    ,("lucille","205-2928")
    ,("wendy","939-8282")
    ,("penny","853-2492")
    ]
```

Ми бачимо, що тип `phoneBook` є `[(String,String)]`. Це нам каже, що ми маємо справу із асоціативним списком, який асоціює рядки із рядками, і більш нічого. Створімо синонім цього типу, щоб надати ще трохи додаткової інформації в оголошенні типу.

```
type PhoneBook = [(String,String)]
```

Тепер оголошенням типу для нашої телефонної книги може бути `phoneBook :: PhoneBook`. Так само створімо синоніми для типу `String`.

```
type PhoneNumber = String
```

```
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

Типи-синоніми до `String` Хаскел-програмісти зазвичай створюють, коли хочуть надати більше інформації про семантичне навантаження цих рядків, і про те, як ці рядки мають використовуватись у функціях.

А тепер, означуючи функцію, яка приймає пару — ім'я та число — і відповідає, чи є така пара у нашій телефонній книзі, ми можемо почати з гарного і інформативного оголошення типу.

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
```

Якщо б ми вирішили не користуватися тут синонімами, ця функція мала б тип `String -> String -> [(String,String)] -> Bool`. Оголошення із синонімами, як бачимо, легше читати і розуміти. Але з синонімами можна як недоборщити, так і переборщити! Синоніми зазвичай означують для того, щоб описати, що саме представляють «стандартні» типи у наших функціях (і таким чином оголошення типів стають краще задокументованими); також синоніми стають в пригоді там, де в оголошенні функцій є «порівняно довгі» типи (на кшталт `[(String,String)]`), що повторюються багато разів і мають якесь специфічне семантичне навантаження в контексті цих функцій.

Синонімічні типи можна параметризувати. Хай нам потрібен тип, який представлятиме асоціативний список і ми хочемо, щоб він був достатньо загальним. Щоб можна було використовувати які завгодно типи в ролі ключів і значень, можемо зробити так:

```
type AssocList k v = [(k,v)]
```

Тепер для функції, яка бере ключ і шукає відповідне тому ключе́ві значення в асоціативному списку, можна записати сиґнатуру `Eq k => k -> AssocList k v -> Maybe v`. Конструктор типу `AssocList` приймає два типи і повертає конкретний тип — такий як, наприклад, `AssocList Int String`.

**«Конкретний тип» каже**: Увага! Коли я кажу *конкретні типи*, я маю на увазі або «повністю застосовані» конструктори типів на кшталт `Map Int String`, або ж мова іде про поліморфні функції — фіговини на кшталт `[a]` або `Ord a => Maybe a` і всяке таке. А ще, деколи я та інші непересічні типи вживаємо такі «розмиті» фрази як «тип `Maybe`», але насправді ми не маємо на увазі «тип», бо кожен чайник знає, що `Maybe` є конструктором типів. Коли я подам

тип `String` до `Maybe`, і отримаю `Maybe String` — ось тоді я матиму конкретний тип! Значення мають тип, і цей тип може бути лише типом конкретним. Отож, підсумовуючи, живи швидко, кохай палко і тримай порох сухим!

Ми можемо будувати нові функції частковим застосуванням інших функцій. За аналогією, частково застосовуючи конструктори типів, ми отримуємо нові конструктори типів. Так само, як виклик функції із недостатньою кількістю параметрів повертає нову функцію, не додаючи параметрів конструкторові типів, ми отримуємо частково застосований конструктор типів. Якщо нам треба тип, який представлятиме усі мапи `Data.Map` із цілих чисел у що завгодно, ми можемо або записати його так:

```
type IntMap v = Map Int v
```

Або ж так:

```
type IntMap = Map Int
```

В обох реалізаціях, конструктор типу `IntMap` приймає один тип-параметр, і цей тип параметризуватиме те, у що саме ця мапа відображатиме цілі числа.

**Мало не забув!** Якщо ви збираєтеся це реалізувати, скоріш за все ви зробите імпорт `Data.Map` в підпростір імен. Якщо ви імпортували в підпростір, ім'я підпростору має передувати й іменам конструкторів типів. Отже, якщо ім'я підпростору є Map (буде таке за замовчуванням, якщо ви виконаєте `import qualified Data.Map`), треба буде переозначити наш `IntMap` отак: `type IntMap = Map.Map Int`.

Впевніться, що ви дійсно розумієте різницю між конструкторами типів і конструкторами значень. Ми створили типи-синоніми `IntMap` чи `AssocList`, які насправді є конструкторами типів (синонімами конструкторів типів!), і їх також не варто плутати із конструкторами значень. Тому речі такі, як, наприклад, `AssocList [(1,2),(4,5),(7,9)]`, є нісенітницями. Все, що дозволяє нам синонімія, — це вживати інше ім'я для типу чи конструктору типів в в сиґнатурі типу функції чи в анотації типу виразу. Ми можемо записати `[(1,2),(3,5),(8,9)] :: AssocList Int Int`, і це означатиме, що числа всередині пар асоціативного списку матимуть тип `Int`, але ми все ще можемо використовувати такий список як звичайний список пар цілих чисел. Типи-синоніми (і типи загалом) можуть лише використовуватися в тих місцях в хаскельному

коді, де йдеться про типи. А саме — в означеннях нових типів (тобто, в означеннях за допомогою ключових слів *data* та *type*) або коли ми знаходимося після `::` (тобто, в типосиґнатурах функцій та в анотаціях типів).

Інший чудовий двопараметровий конструктор типів — `Either a b`. Орієнтовно він є означений ось так:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Він має два конструктори значень. Конструктор `Left` містить щось типу `a`, а `Right` — щось типу `b`. Тому можна використовувати цей тип для інкапсуляції значення якогось одного типу або значення якогось іншого типу. Коли ми отримуємо значення типу `Either a b`, ми зазвичай зіставляємо із взірцем по кожному з конструкторів і далі робимо різні речі, в залежності від того, зіставився конструктор `Left` чи `Right`.

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

Ми бачили, що `Maybe a` використовувався в основному для представлення результатів обрахунків, які могли або успішно, або неуспішно завершитися. Але іноді `Maybe a` недостатньо, оскільки `Nothing` не передає достатньо інформації про природу того неуспіху (все, що передається — це те, що обрахунок був неуспішним). Це підходить, наприклад, для функцій, які можуть «неуспішно завершитися» тільки в один спосіб. Або ж для випадків коли нас не цікавить причина неуспіху. До прикладу: пошук, який виконується `Data.Map.lookup`, не завершиться поверненням значення лише якщо мапа не містить ключа, за яким ведеться пошук, а отже ми знаємо точно що сталося. Однак якщо нас цікавить, чому якась функція завалилася, або ж в який спосіб з декількох можливих вона завалилася, то ми зазвичай повертаємо з такої функції `Either a b`, де `a` є типом, що звітує нам про можливу причину неуспіху, а `b` є типом значення-результату, який (значення-результат) повертається у випадку успішного завершення. Отже, помилки використовують конструктор значень `Left`, тоді як результати використовують `Right`.

Приклад: в школі є шафки для того, щоб школярі мали куди покласти свої плакати гурту Guns'n'Roses. Кожна шафка має замок, а кожен замок має **код** [code combination]. Коли школяр хоче отримати шафку, він каже вчителеві її номер, а той видає код замка для неї. Однак, якщо шафка вже використовується

кимось іншим, вчитель не може видати її код, і тоді потрібно обрати якусь іншу. Скористаємося `Data.Map`, щоб представити шафки. Номер шафки буде відображатися в пару значень — стан шафки (зайнята чи вільна) і код замка.

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

Прості речі. Ми означили новий тип даних, що описує, чи є вільною шафка чи ні, і створили тип-синонім для коду замка. Ми також означили мапу шафок як тип-синонім до типу, що відображає цілі числа в пари `(LockerState, Code)`. А тепер ми напишемо функцію, яка шукає код у мапі шафок. Ми скористаємося типом `Either String Code` для представлення нашого результату, бо ми можемо і не знайти код. Причини дві — шафка може бути зайнятою і в цьому випадку ми не маємо права показати код, або ж шафки із таким номером взагалі не існує. Якщо пошук завершився неуспішно, ми просто скористаємося `String` для повідомлення, що саме трапилося.

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
    case Map.lookup lockerNumber map of
        Nothing -> Left $ "Locker number " ++ show lockerNumber
                                           ++ " doesn't exist!"
        Just (state, code) -> if state /= Taken
                              then Right code
                              else Left $ "Locker " ++ show lockerNumber
                                                    ++ " is already taken!"
```

Ми виконуємо звичайний пошук в мапі. Якщо ми отримуємо `Nothing`, то повертаємо значення типу `Left String`, яке каже, що шафка взагалі не існує. Якщо ж ми знаходимо таку шафку, тоді виконуємо додаткову перевірку, щоб дізнатися чи зайнята ця шафка. Якщо так, то повертаємо `Left` з повідомленням про те, що вона вже зайнята. Якщо не зайнята, тоді повертаємо значення типу `Right Code`, в якому ми даємо учневі правильний код замка. Насправді типом результату є `Right String`, але ми ввели синонім до нього, щоб надати трохи більше інформації про цей тип в сиґнатурах типу. Ось приклад мапи:

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken,"ZD39I"))
```

```
    ,(101,(Free,"JAH3I"))
    ,(103,(Free,"IQSA9"))
    ,(105,(Free,"QOTSA"))
    ,(109,(Taken,"893JJ"))
    ,(110,(Taken,"99292"))
    ]
```

Тепер спробуймо пошукати там якісь коди замків.

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

Ми могли б скористатися `Maybe a` для представлення результату, але тоді не змогли б дізнатися у випадку неуспішного пошуку, чому не отримали код. А в теперішній реалізації маємо інформацію про причину неуспіху — її вбудовано в тип-результат.

## 8.6 Рекурсивні структури даних

Як ми вже бачили, конструктор в алгебраїчному типі даних може мати кілька полів (а може й не мати взагалі) і кожне поле повинно бути якогось конкретного типу. Цікаво, що ми також можемо створювати типи, в яких є конструктори, що мають поля такого ж типу, як і тип, що ми його створюємо! Іншими словами, ми можемо означувати рекурсивні типи, де одне значення якогось типу містить значення того ж типу, яке, в свою чергу, містить ще одне або декілька значень того ж самого типу і так далі.

Поміркуймо над цим списком: `[5]`. Це просто синтаксичний цукор для `5:[]`. Ліворуч від `:` стоїть значення, а праворуч — список, і в цьому випадку він порожній. Тепер, а як щодо списку `[4,5]`? Що ж, це розцукровується в `4:(5:[])`. Дивимося на перший оператор `:` і бачимо, що праворуч стоїть елемент, а ліворуч — список `5:[]`. Так само можна «розібрати

по запчастинах» список `3:(4:(5:6:[]))` — його можна записати як `3:4:5:6:[]` (бо оператор `:` є правоасоціативним) або ж як `[3,4,5,6]`.

Можна сказати, що список може бути порожнім списком або він може бути елементом, з'єднаним за допомогою `:` із іншим списком (який може бути порожнім списком або не бути).

Тож скористаймося алгебраїчними типами даних для створення нашого власного списку!

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

Це читається так само, як і наше означення списків у одному із попередніх абзаців. Список є або порожнім списком, або є поєднанням значення і списку. Якщо це збиває з пантелику, можливо буде легше зрозуміти це означення, якщо його переписати із використанням синтаксису для записів?

```
data List a = Empty |
    Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord)
```

Можливо вас також підбентежує[*] конструктор `Cons`? *cons* — це словосинонім до `:`. Річ у тім, що в списках `:` є насправді конструктором, який приймає значення та інший список і повертає список. Ми вже можемо тут скористатися нашим новим типом для списків! Іншими словами, конструктор `:` має два поля. Одне поле типу `a`, а інше — типу `[a]`.

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

Ми викликали наш конструктор `Cons` в інфіксний спосіб, щоб можна було легше бачити його подібність до `:`. Конструктор `Empty` — схожий на `[]`, а вираз `4 `Cons` (5 `Cons` Empty)` — подібний до `4:(5:[])`.

Якщо назва функції складається лише з спецсимволів (на кшталт `>`, `:`, `$`, `+` і таке подібне), ця функція стає інфіксною автоматично[‡]. Так само із конструкторами[§]: вони ж бо є звичайними функціями, які повертають значення.

[*] Це слово є ©Олег Науменко.

[‡] Можливо, треба буде увімкнути LANGUAGE-директиву `TypeOperators`. Також, ім'я інфіксної функції не може починатися з `:`.

[§] Але є обмеження: ім'я конструктора значень має починатися з `:`.

Дивіться!

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

Перш за все, зверніть увагу на нову синтаксичну конструкцію для **оголошення асоціативності** [fixity declaration]. Коли ми означуємо функції як оператори, ми можемо вказати для них асоціативність (але це не обов'язково). Асоціативність описує, яким є оператор — право- чи **лівоасоціативним** [left-associative] і наскільки «сильно». Наприклад, оголошення асоціативності для `*` є `infixl 7 *`, а для `+` — `infixl 6 +`. Це означає, що вони обидва є лівоасоціативними (тобто, `4 * 3 * 2` — це є `(4 * 3) * 2`), але `*` **зв'язує сильніше** [binds tighter], ніж `+`, бо має більше значення асоціативності (7 в `*` проти 6-ти в `+`), і тому `5 + 4 * 3` — є це `5 + (4 * 3)`.

Інше, що ми змінили — ми просто записали `a :-: (List a)` замість `Cons a (List a)`. Тепер ми можемо складати списки нашого найновішого спискового типу ось так:

```
ghci> 3 :-: 4 :-: 5 :-: Empty
(:-:) 3 ((:-:) 4 ((:-:) 5 Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Empty)))
```

Оскільки ми автоматично втілили `Show` для нашого типу, Хаскел покаже його, але інтерпретуватиме наш конструктор як префіксну функцію — ось звідки дужки навколо оператора (пам'ятайте, що `4 + 3` — це є `(+) 4 3`).

Побудуймо функцію, яка додає два списки в один. Ось як означений оператор `++` для звичайних списків.

```
infixr 5  ++
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

То ми просто поцупимо це означення для нашого власного списку. Назвемо функцію `.++`.

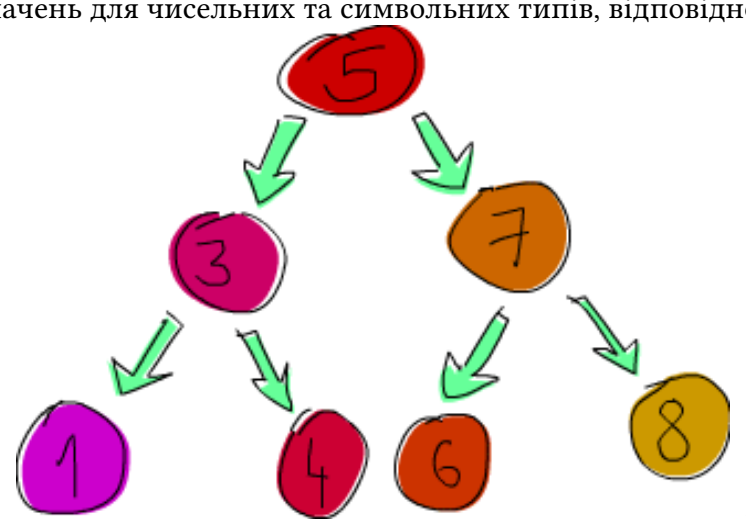
```
infixr 5  .++
(.++) :: List a -> List a -> List a
Empty .++ ys      = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)
```

І перевірмо, чи працює...

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Empty))))
```

Гарно. Є гарно. Якщо б захотіли, ми могли б втілити всі функції, що працюють із списками, для нашого власного спискового типу.

Зверніть увагу, як ми зіставили із взірцем `(x :-: xs)`. Зіставлення із взірцем в Хаскелі насправді зіставляє із взірцем конструктори. Ми можемо зіставити `:-:` тому, що це є конструктор значень для нашого власного списку, і можемо також зіставити `:`, бо це є конструктор значень для стандартного списку. Те ж стосується й `[]`. Оскільки зіставлення з взірцем «іде» (лише) по конструкторах, ми здатні зіставляти всякі такі штукенції — звичайні там префіксні конструктори, чи то речі такі як `8` або `'a'`, які є, по суті, конструкторами значень для чисельних та символьних типів, відповідно.




Зараз ми запрограмуємо **бінарне дерево пошуку [binary search tree]**. Якщо ви не знайомі із бінарними деревами (пошуку) з мов таких як C, ось, коротко, про цю структуру даних: дерево складається з елементів (або вузлів); зазвичай кожен елемент дерева містить в собі значення і, на додачу, ще й вказує на два інші елементи (так звані *елементи-діти*); дітей розрізняють — є ліве дитинча, і є дитинча праве. Значення в лівій дитині є меншим за значення елемента-батька, а в правій — більшим (а якщо коротко — лівий елемент є меншим за батьківський, а правий — більшим). Кожен із елементів-дітей також може вказувати ще на два елементи. В загальному випадку дітей може й не бути, або може бути лише одна дитина. В результаті, кожен елемент може мати не більше двох піддерев. Найцікавішою властивістю, що випливає з такої побудови, є те, що відомо наперед, що всі елементи в лівому піддереві елемента, в якого, скажімо, значення є п'ятірка, будуть меншими за п'ять. А елементи в правому піддереві будуть більшими.

Якщо нам потрібно перевірити, чи 8 є в нашому дереві (див. малюнок), ми починаємо з 5, і, оскільки 8 більше за 5, йдемо праворуч. Тепер ми опинилися біля елементу із сімкою, і, оскільки 8 більше за 7, ми знову ідемо праворуч. Ура — ми знайшли наше число, за три кроки! Якби ж це був звичайний список (або дерево, але не збалансоване), нам би довелося здійснити сім кроків замість трьох, щоб перевірити чи є там вісімка.

Множини і мапи із `Data.Set` та `Data.Map` реалізовані на базі дерев, тільки замість звичайних бінарних дерев пошуку вони використовують збалансовані бінарні дерева, які завжди себе підтримують в хорошій (збалансованій) формі. Але зараз ми будемо реалізовувати звичайні бінарні дерева пошуку.

А тепер проговорюються отакі слова[†]: дерево є або порожнім, або ж воно є елементом, який містить якесь значення і два піддерева. Схоже, в пригоді стане алгебраїчний тип даних!

```
data Tree a = EmptyTree
            | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

Гаразд, добре, то є добре. Замість того, щоб вручну будувати дерево, ми напишемо функцію, яка бере дерево та значення і вбудовує у те дерево вузол із тим значенням. Досягається це ось як: порівнюємо значення, яке хочемо вставити, із значенням у кореневому вузлі, і, якщо воно менше за корінь, йдемо наліво, а якщо більше — направо. Ми виконуємо цю процедуру для кожного наступного вузла, допоки досягнемо порожнього дерева. Як тільки його досягли, просто замінюємо те порожнє дерево на вузол із нашим значенням.

В мовах на кшталт C ця процедура була б реалізована за допомогою змін **вказівників** [pointers] та змін значень всередині вузлів дерева. В Хаскелі ж, насправді, ми не можемо змінити дерево, тому нам потрібно створювати нове піддерево щоразу, коли було прийняте рішення про те, куди треба йти (вліво або вправо). **Функція вставки** [insert function] повертає зовсім[‡] нове дерево, оскільки в Хаскелі немає концепції вказівника — Хаскел знає лише про **значення** [values]. Таким чином, тип нашої функції вставки буде схожим на `a -> Tree a -> Tree a`. Функція бере значення і дерево та повертає нове дерево, в яке вбудовано новий елемент із цим значенням. Цей підхід може здатися **неефективним** [inefficient], але він є таким, бо **чистофункційні структури даних** [purely functional data structures] є **незміненними** [immutable], а це дозволяє реалізації, де стара (до модифікації) і нова (після) версії структури даних мають (через так званий **поділ** [sharing]) багато спільних частин[†].

[†]Фраза «А тепер проговорюються отакі слова» є ©Анісімов Ігор Олексійович. Використано без дозволу.

[‡]Семантично «зовсім нове». Але реалізація зазвичай така, що воно матиме спільні частини із попереднім деревом.

[†]Легше перепросити, ніж переписати це речення. Отже — перепрошую! :) NB: Тепер зро-

Отже, ось дві функції. Одна є **допоміжною функцією** [utility function] для створення дерева із одним-єдиним вузлом — так званого **однодерева** [singleton tree], а інша — для вставки значення в дерево.

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: Ord a => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

Функція `singleton` є допоміжною функцією для створення нашвидкоруч вузла, який має усередині якесь значення і два порожні піддерева. У функції вставки ми спочатку перевіряємо взірцем крайову умову. Якщо ми досягли порожнього піддерева, значить ми є там, де ми хочемо бути, і тоді ми замінюємо те порожнє дерево на однодерево із нашим значенням. Якщо ми вставляємо в **непорожнє дерево** [non-empty tree], ми повинні дещо перевірити. Перш за все, якщо значення, яке ми вставляємо, дорівнює кореневому, просто повертаємо те дерево без змін. Якщо менше, повертаємо дерево, що є схожим на дерево на вході: значення в корені без змін, праве піддерево без змін, але замість лівого піддерева ми вбудовуємо в нього нове піддерево, в яке (буде) вставлено наше значення. Те ж саме (з точністю до заміни ліво на право) відбувається, якщо наше значення є більшим за значення з **кореневого елемента** [root node].

Далі ми створимо функцію, яка перевіряє, чи містить дерево певний елемент. По-перше, означмо **крайову умову** [edge condition]. Якщо ми шукаємо елемент у порожньому дереві, то його там явно немає. Гаразд. Зауважте, як це схоже на крайову умову пошуку елементів у списках. Шукати елемент у порожньому списку теж не варто, бо його там немає (і не тільки його — там немає багатьох інших елементів!). Порожнеча… Але годі з цим, рухаємося далі: якщо ми шукаємо елемент в непорожньому дереві, тоді нам треба дещо перевірити. Якщо значення елемента в кореневому вузлі дорівнює значенню, що ми шукаємо — ура! А якщо ні — що тоді? Тоді ми можемо використати той факт, що всі «ліві елементи» є менші за кореневий. Тому, якщо елемент, що ми шукаємо, є менший за кореневий, тоді ми продовжуємо пошуки у лівому піддереві. Якщо ж він більший — будемо шукати його у правому.

```
treeElem :: Ord a => a -> Tree a -> Bool
treeElem x EmptyTree = False
```

зуміло, чому цього речення бракує в оригіналі.

```
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right
```

Як бачимо, параграф слів втілився в п'ять рядків хаскелівського коду. Пограймося-но із нашими деревами! Замість того, щоб будувати дерево вручну (хоча ми б могли), ми скористаємося згортком і будуватимемо дерево зі списку. Пам'ятайте: багато алгоритмів, в яких список обробляється елемент за елементом і які повертають якесь значення, можна реалізувати за допомогою згортка! Ми почнемо із порожнього дерева і згортатимемо список з правого боку, вставляючи елемент за елементом в наше дерево-накопичувач.

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5 (Node 3 (Node 1 EmptyTree EmptyTree)
 (Node 4 EmptyTree EmptyTree))
  (Node 7 (Node 6 EmptyTree EmptyTree)
   (Node 8 EmptyTree EmptyTree))
```

`treeInsert` у виклику `foldr` була **згортаючою функцією** [folding function] (вона бере дерево та елемент зі списку та повертає нове дерево), а `EmptyTree` був стартовим накопичувачем. `nums`, звісно, був списком, який ми згортали.

Коли ми друкуємо те дерево в консолі, воно не дуже легко читається, але якщо постараємося, можемо розгледіти його топологію. Бачимо, що кореневий елемент містить 5 і має два піддерева, із значеннями 3 та 7 в кореневих елементах цих двох піддерев, і так далі.

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

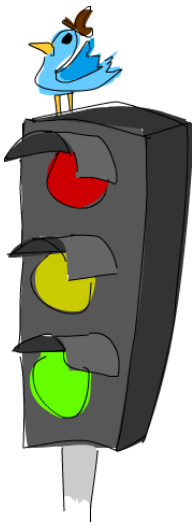
**Перевірка на наявність** [membership check] елемента також гарненько працює. Чудово.

Отже, як ми побачили, алгебраїчні структури даних є дійсно крутими й потужними концепціями у Хаскелі. Ми можемо використовувати їх в побудові будь-чого, починаючи від булевих значень та **перелічення днів тижня**

[weekday enumeration] і не закінчуючи бінарними деревами пошуку і багато чим іще!

## 8.7 Вступ до типокласів

Наразі ми вивчили деякі стандартні типокласи Хаскела і розглянули, які до них належать типи. Ми також навчилися автоматично створювати втілення стандартних типокласів для наших власних типів — себто, не писали втілення самі, а просили, щоб Хаскел вивів їх для нас. Ну а у цьому розділі ми навчимося створювати нові, власні типокласи і втілювати їх вручну.



Коротко нагадаємо про типокласи: типокласи схожі на інтерфейси. Типоклас означує деяку поведінку (як-от перевірку на рівність, порівняння для впорядкування, можливість перелічення), а потім для типів, які можуть поводитися таким чином, пишуться відповідні втілення. Втілити типоклас — це те саме, що означити функції, означування яких той типоклас «вимагає». Тому, коли ми кажемо, що «наш тип є втіленням типокласу» або «наш тип втілює типоклас», то маємо на увазі, що ми можемо використовувати функції, типосиґнатури яких той типоклас означує[†], в роботі із цим нашим типом.

Типокласи не мають практично нічого спільного із класами в мовах таких як Java або Python. Це заплутує багатьох людей, тому я б хотів, щоб ви прямо зараз забули все, що знали про класи в імперативних мовах.

Наприклад, типоклас `Eq` є для речей, які можна перевірити на рівність — і він вимагає означення функцій `==` та `/=`. Якщо ми маємо якийсь тип (скажімо `Car`), а перевірка двох машин на рівність є цілком природною (за допомогою функції `==`), тоді є цілком природно зробити тип `Car` членом `Eq`, написавши відповідне втілення.

Ось як типоклас `Eq` є означено у стандартному модулі `Prelude`:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Чекайте, чекайте, чекайте! Якийсь новий дивний синтаксис та нові ключові слова! Не хвилюйтеся, за секунду все стане зрозумілим. Спершу, коли ми пишемо `class Eq a where`, це означає, що ми означуємо новий типоклас, який зветься `Eq`. `a` є параметром типу і це означає, що `a` гратиме роль типу, який втілюватиме `Eq`. Він не обов'язково повинен називатися `a`, ім'я не обов'язково

[†]І, в такий спосіб, вимагає означення цих функцій від членів. Але типоклас може також пропонувати й означення для цих функцій.

має складатися із однієї літери — головне, щоб те ім'я починалося з малої літери. Тоді ми означаємо декілька функцій. Надання означень не є обов'язковим — вимагається лише надати типосиґнатури для функцій (себто, тип функції і її ім'я).

**Примітка**: Декому могло бути зрозумілішим, якби ми замість `a` вжили б щось більш інформативне, як от, наприклад, `equatable` (з англійської, «equatable» — «той, кого можна перевірити на рівність»), і записали `class Eq equatable where`, а потім оголосили б тип отак: `(==) :: equatable -> equatable -> Bool`.

А втім, ми *таки записали* тіла для функцій, яких вимагає `Eq`, тільки от ми означили їх взаєморекурсивно. Ми сказали, що два значення типокласу `Eq` є рівними, якщо вони не різні, і що вони різні, якщо вони не дорівнюють одне одному. Насправді, це робити було зовсім не обов'язково, але ми зробили — і скоро побачимо, як нам це стане в пригоді.

**Примітка**: Якщо ми напишемо `class Eq a where`, а потім подамо оголошення типу «усередині» цього типокласу на кшталт `(==) :: a -> a -> Bool`, то в майбутньому, якщо перевіримо тип такої функції, він буде `Eq a => a -> a -> Bool`.

Тепер в нас є типоклас — і що ми можемо з ним робити? Що ж, зовсім небагато, насправді. Але тільки-но почнемо будувати втілення того типокласу, почнемо отримувати гарний **функціонал** [functionality]. Тож розгляньмо наступний тип:

```
data TrafficLight = Red | Yellow | Green
```

Він означує стани світлофора. Зауважте, що цього разу ми не втілювали ніякі типокласи автоматично. Це тому, що ми плануємо написати декілька втілень вручну, хоча типокласи такі як `Eq` та `Show` ми могли б втілити автоматично. Ось як `TrafficLight` втілюватиме `Eq`:

```
instance Eq TrafficLight where
    Red == Red       = True
    Green == Green   = True
    Yellow == Yellow = True
    _ == _           = False
```

Тут ми скористалися **ключовим словом** [keyword] *instance*. Таким чином, ключове слово *class* призначене для означування нових типокласів, а *instance* — для створення втілень типокласів для наших типів. Коли ми означували `Eq`, то записали `class Eq a where` і сказали, що `a` є тип, втілення якого буде створено пізніше, і під `a` розумівся якийсь (будь-який) тип. Можемо легко тут це бачити, бо, коли створюємо втілення, пишемо `instance Eq TrafficLight where` — замінюємо `a` на тип, для якого пишемо втілення, — себто, на `TrafficLight`.

Оскільки `==` було означено в типокласі `Eq` **через** [in terms of] `/=` та навпаки, тепер нам треба було лише **замістити** [to override] якесь одне із двох у втіленні. Це зветься **мінімальним повним означенням** [minimal complete definition] для типокласу, себто, — мінімальним набором функцій, які потрібно реалізувати для типу, щоб він міг поводитися так, як зазначає типоклас. Мінімальне повне означення для `Eq` — це або заміщення `==`, або заміщення `/=`. Якщо б `Eq` було означено так:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

нам би довелося писати тіла для обох цих функцій у кожному втіленні, бо Хаскел не знає, що ці функції пов'язані. Тоді мінімальним повним означенням був би боєкомплект з `==` та `/=`.

Як бачите, ми реалізували `==` просто за допомогою зіставлення із взірцем. Оскільки випадків, де два сигнали світлофора є нерівними, є набагато більше, ми вказали лише ті, коли сигнали однакові, а потім просто зловили решту у фінальний **універсальний взірець** [catch-all pattern] — якщо це не одна із попередніх комбінацій, то сигнали не однакові.

Втілимо також вручну типоклас `Show`. Щоб задовольнити вимоги мінімального повного означення для `Show`, потрібно просто реалізувати його функцію `show`, яка приймає значення і перетворює його на рядок.

```
instance Show TrafficLight where
    show Red    = "Red light"
    show Yellow = "Yellow light"
    show Green  = "Green light"
```

Знову ж таки, ми скористалися зіставленням із взірцем для досягнення мети. Погляньмо на це в дії:

```
ghci> Red == Red
True
ghci> Red == Yellow
```

```
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Симпатично. Ми могли б автовтілити `Eq` і це було б повним еквівалентом цієї нашої «ручної роботи» (але цього разу ми реалізували вручну з педагогічних міркувань). Однак, автовтілення `Show` перетворювало б конструктори значень на рядки безпосередньо. А тут нам закортіло, щоб сигнали серіалізувалися в рядки шаленої інформативності, як от `"Red light"`, і тому нам довелося втілювати вручну.

Можна також створювати типокласи, які є **підкласами** [subclasses] інших типокласів. Означення типокласу `Num` є доволі довгим, але ось як виглядає його початок:

```
class Eq a => Num a where
   ...
```

Як ми вже згадували, є багато місць, куди можемо **увіпхнути** [cram in] умови типокласів. І тут — одне з таких місць: `class Eq a => Num a where` — це є те ж саме що й `class Num a where`, тільки ми вимагаємо, що наш тип `a` повинен втілювати `Eq`. Ми, по суті, кажемо, що потрібно втілити `Eq` до того, як можна починати втілювати `Num`. Перед тим, як якийсь тип зможе вважатися числом, цілком природно вимагати можливості перевірки на рівність для значень цього типу. От і власне все, що треба сказати про створення **підкласів** [subclass], — це є просто додавання **умови типокласу** [class constraint] в означення типокласу. І тепер, коли ми означуємо тіла функцій у *class*-означенні чи то коли ми означуємо їх у *instance*-означенні, ми спираємося на те, що `a` є членом `Eq` і, таким чином, можна користуватися `==` і перевіряти на рівність значення, що мають тип `a`.

А як же тип `Maybe` та списковий тип — чи можуть вони втілювати типокласи? Те, що відрізняє `Maybe` від, скажімо, `TrafficLight` — це те, що `Maybe` не є конкретним типом, а конструктором типу, який приймає один тип-параметр (наприклад, `Char` чи щось інше) і повертає **конкретний тип** [concrete type] (наприклад, `Maybe Char`). Погляньмо на типоклас `Eq` знову:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

З оголошень типу бачимо, що `a` використовується як конкретний тип, бо всі типи у функціях повинні бути конкретними (пам'ятайте, ви не можете написати функцію, що має тип `a -> Maybe`, але можете написати функцію, що має тип `a -> Maybe a` чи тип `Maybe Int -> Maybe String`). Ось чому ми не можемо записати щось на кшталт

```
instance Eq Maybe where
    ...
```

бо, повторюючись, `a` має бути конкретним типом, але `Maybe` не є конкретним типом — це конструктор типу, який приймає один параметр та повертає конкретний тип. Але було б дуже нудно писати `instance Eq (Maybe Int) where`, `instance Eq (Maybe Char) where` і так далі для всіх існуючих в світі типів! Замість цього можна записати це ось як:

```
instance Eq (Maybe m) where
    Just x == Just y   = x == y
    Nothing == Nothing = True
    _ == _             = False
```

Це є те ж саме, що сказати, що ми хочемо зробити всі типи «на зразок» `Maybe <<something>>` членами `Eq`. Насправді ми могли б так і назвати той параметр **«something»** [«щось»], і записати `(Maybe <<something>>)`, але зазвичай, коли ми іменуємо параметри типів, ми надаємо перевагу ім'ям з однієї літери, залишаючись вірними стилю Хаскела. `(Maybe m)` грає роль `a` в `class Eq a where`. У той час як `Maybe` не є конкретним типом, `Maybe m` ним є. Вказавши параметр типу (себто, `m`, мала літера), ми сказали, що хочемо, щоб всі типи на зразок `Maybe m`, де `m` є будь-яким типом, були членами типокласу `Eq`.

Але тут є одна проблемка. Ви бачите її? Ми застосовуємо `==` до «вмісту» `Maybe`, не маючи жодної гарантії, що той вміст є членом `Eq`! Ось чому нам потрібно змінити наше *instance*-означення таким от чином:

```
instance Eq m => Eq (Maybe m) where
    Just x == Just y   = x == y
    Nothing == Nothing = True
    _ == _             = False
```

Нам потрібно було додати умову типокласу! У цьому *instance*-означенні ми кажемо: ми хочемо, щоб всі типи зразка `Maybe m` були членами типокласу `Eq`, але тільки ті типи, в яких `m` (тобто, вміст `Maybe`) є членом `Eq`. Автоматичне втілення виглядало б точнісінько так.

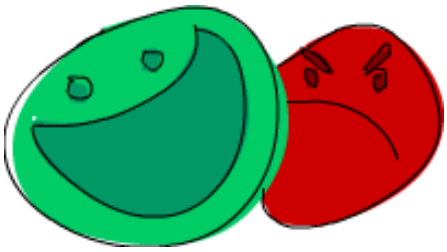
У більшості випадків умови типокласу в *class*-означеннях використовуються для того, щоб зробити типоклас підкласом іншого типокласу, а умови типокласу в *instance*-означеннях використовуюся для того, щоб викласти вимоги до типів, що параметризують тип, для якого пишеться втілення. Наприклад, тут ми вимагали, щоб вміст `Maybe` був членом типокласу `Eq`.

Якщо, втілюючи типоклас, ви бачите, що якийсь тип використовується в оголошенні як конкретний (як, наприклад, `a` в `a -> a -> Bool`), ви маєте подати всі параметри типу, щоб отримався конкретний тип, і взяти усе це діло в дужки.

**Примітка**: Зверніть увагу, що тип, для якого ви пишете втілення якогось типокласу, замінить параметр у *class*-означенні того типокласу. `a` із `class Eq a where` буде замінено справжнім типом, коли писатиметься втілення, тому намагайтеся подумки розмістити ваш тип також і в оголошеннях типів функцій з відповідного типокласу. `(==) :: Maybe -> Maybe -> Bool` — нісенітниця, тоді як `(==) :: Eq m => Maybe m -> Maybe m -> Bool` можна зрозуміти. Але це — лише інформація для роздумів, не більше, бо `==` завжди матиме тип `(==) :: Eq a => a -> a -> Bool`, незалежно від того, скільки втілень `Eq` ми напишемо.

О, ще одне! Якщо ви хочете побачити всі втілення якогось типокласу, просто спитайте `:info YourTypeClass` у GHCi. Наприклад, `:info Num` спочатку покаже список функцій, яких вимагає типоклас `Num`, а після того — список усіх членів цього типокласу! `:info` також працює із типами і конструкторами типів. Якщо ви запитаєте `:info Maybe`, вам буде показано всі типокласи, які `Maybe` втілює. А ще `:info` хрумає функції і показує їх оголошення типу. Я вважаю, що це — суперкруто.

## 8.8 Типоклас «так-ні»



У JavaScript-і та деяких інших **слабкотипізованих мовах** [weakly typed languages] ви можете записати практично будь-що у **вираз розгалуження** [if expression]. Наприклад, всі оці речі є дозволеними:

- `if (0) alert(”YEAH!”) else alert(”NO!”)`,
- `if (””) alert(”YEAH!”) else alert(”NO!”)`,
- `if (false) alert(”YEAH!”) else alert(”NO!”)`,

тощо. І в кожному з цих випадків буде викинуто віконце-попередження із `NO!`. Якщо ви виконаєте `if (”WHAT”) alert (”YEAH!”) else alert(”NO!”)`, буде викинуто попередження `”YEAH!”`, бо в JavaScript-і непорожні рядки вважаються ближчими до **істинних булевих значень** [boolean values of true] ніж до хибних.

Хоча в Хаскелі краще використовувати в **булевій семантиці** [boolean semantics] виключно булеві значення, спробуймо все одно реалізувати таку JavaScript-ову поведінку. Заради забави! Розпочнімо із *class*-означення.

```
class YesNo a where
    yesno :: a -> Bool
```

Доволі просто. Типоклас `YesNo` оголошує одну функцію. Ця функція приймає одне значення. Потрібно, щоб тип того значення можна було якось пов’язати із поняттям «істинності». І ця функція каже нам вже точно, чи є її аргумент істинним чи хибним. Зверніть увагу, що з того, як ми використовуємо `a` в оголошенні, випливає, що `a` має бути конкретним типом.

Далі означмо деякі втілення. Для чисел ми вважатимемо, що будь-яке число, що не є нулем, є істинним, а нуль — хибним (як і в JavaScript-і).

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

Порожні списки (а тому і порожні рядки, звичайно ж) є більш хибними ніж істинними, тоді як непорожні списки — більш істинними ніж хибними.

```
instance YesNo [a] where
    yesno [] = False
    yesno _  = True
```

Зверніть увагу на те, як ми «просто, лєгко і нєпрінуждьонно» вставили в список параметр `a`, щоб зробити його конкретним типом, навіть якщо ми не робили ніяких припущень щодо природи типу `a`. Що іще, гм… О! Ідея! Є іще, власне, `Bool` і там і так ясно, яке значення є більш істинним, а яке — менше.

```
instance YesNo Bool where
    yesno = id
```

Га? Що в біса таке те `id`? Це стандартна бібліотечна функція, яка бере параметр і повертає його ж, — себто, вона є якраз тим, що нам тут і потрібно було записати.

Напишімо втілення й для `Maybe a`!

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing  = False
```

Знову ж таки, нам не потрібна була тут умова типокласу, бо ми не робили ніяких припущень щодо вмісту `Maybe`. Ми лише сказали, що значення є близьким до істини, якщо воно є `Just`-значення, а близьким до хиби — якщо `Nothing`. Ми все ж повинні були записати `(Maybe a)` замість просто `Maybe`, бо, якщо подумати над цим, функція `Maybe -> Bool` не може існувати (тому що `Maybe` не є конкретним типом), тоді як із `Maybe a -> Bool` все тіп-топ. А найкрутішим наслідком цього є те, що тепер будь-який тип зразка `Maybe <<something>>` є членом типокласу `YesNo` і немає значення, чим є те `<<something>>`.

Раніше ми означили тип `Tree a`, який був реалізацією бінарного дерева пошуку. Ми можемо покласти, що порожнє дерево є хибним значенням, а непорожнє — істинним.

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _         = True
```

Чи можна відобразити значення світлофора в множину значень «так-ні»? Звісно. На червоне світло ви зупиняєтеся. На зелене — рухаєтеся. Якщо жовте? Ех, я зазвичай іду на жовте, бо живу задля адреналіну.

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _   = True
```

Круто, ми понавтілювали собі трохи втілень, — а тепер ходімо пограймося!

```
ghci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
```

```
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: YesNo a => a -> Bool
```

Добре, все працює! Створімо функцію, яка імітує **інструкцію розгалуження** [if statement], але працює зі значеннями типу `YesNo`.

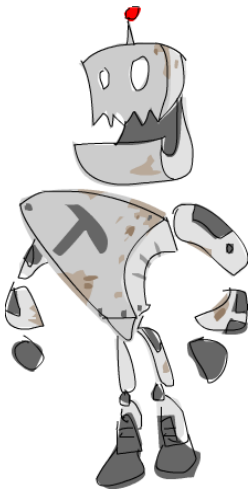
```
yesnoIf :: YesNo y => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal then yesResult else noResult
```

Доволі просто. Функція бере **ніби-«так-ні»-якесь значення** [yes-no-ish value] та дві інші речі. Якщо ніби-«так-ні»-якесь значення є схоже на «так», функція повертає першу із двох речей, інакше повертає другу річ.

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```

## 8.9 Типоклас Functor

Досі нам траплялося багато типокласів зі стандартної бібліотеки. Ми гралися із `Ord`, який призначений для речей, які можуть бути впорядкованими. Ми потоваришували із `Eq`, який є для речей, які можна перевіряти на рівність. Ми побачили `Show`, який є інтерфейсом для типів, значення яких можуть бути перетворені на рядки. Наш хороший друг, `Read`, завше поруч, коли нам потрібно пере-



творити рядок на значення деякого типу. А тепер ми поглянемо на типоклас `Functor`, який, по суті, є призначений для речей, які можна **відображати** [map over]. Ймовірно, ви зараз думаєте про списки, оскільки **відображення списків** [mapping over lists] є **провідною ідіомою** [dominant idiom] в Хаскелі. І ви праві — **тип списків** [list type] є членом типокласу `Functor`.

Чи є кращий спосіб запізнатися із типокласом `Functor`, ніж подивитися на його реалізацію? Підгляньмо!

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Гаразд. Ми бачимо, що він означує одну функцію, `fmap`, і не надає їй **реалізації за замовчуванням** [default implementation]. Тип `fmap` є цікавим. Дотепер у означеннях типокласів змінна типу, яка грала роль типу, для якого означується поведінка, іменувала **конкретний тип** [concrete type], як-от `a` у `(==) :: Eq a => a -> a -> Bool`. Але тепер `f` не є конкретним типом (тобто, не є типом, для якого ми можемо побудувати значення, як-от, наприклад, `Int`, `Bool` чи `Maybe String`), а є конструктором типу, який бере один тип-параметр. Короткий приклад-освіжувач пам'яті: `Maybe Int` є конкретним типом, але `Maybe` є конструктором типу, який приймає один тип як параметр. В усякому разі, ми бачимо, що `fmap` приймає як перший параметр функцію, що перетворює один тип на інший, як другий — функтор, який є застосовано до одного типу, а повертає як результат функтор, застосований до іншого типу.

Якщо це звучить спантеличуюче — не хвилюйтеся. Все стане на свої місця незабаром, коли ми розглянемо декілька прикладів. Хмм, це оголошення типу для `fmap` щось мені нагадує. Якщо ви не знаєте, яка сиґнатура типу в функції `map`, ось вона: `map :: (a -> b) -> [a] -> [b]`.

О, як цікаво! `map` приймає функцію, яка перетворює один тип в інший, та список елементів одного типу і повертає список елементів іншого типу. Друзі, здається ми маємо собі функтор! Справді, `map` є просто `fmap`, яка працює лише зі списками. Ось як список втілює типоклас `Functor`.

```
instance Functor [] where
    fmap = map
```

Ось і все! Зауважте, ми не написали `instance Functor [a] where`, бо із `fmap :: (a -> b) -> f a -> f b` бачимо, що `f` має бути конструктором типу, який приймає один тип-параметр. `[a]` уже є конкретним типом (список із елементів типу `a`) тоді, як `[]` є конструктором типів, який приймає один тип і

може будувати нам такі типи як `[Int]`, `[String]` чи навіть `[[String]]`.

Оскільки для списків `fmap`-ом є просто функція `map`, ми отримуємо однакові результати в обох випадках:

```
map :: (a -> b) -> [a] -> [b]
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

Що ж трапиться, якщо ми застосуємо `map` або `fmap` до порожнього списку? Ну звісно ми отримаємо порожній список. Вони просто перетворюють порожній список типу `[a]` в порожній список типу `[b]`.

Типи, які можуть поводитися як **коробки** [boxes], можуть бути функторами. Уявіть собі, що список є коробкою, яка має нескінченну кількість маленьких відділень, всі з яких можуть бути порожніми, або одне-єдине може бути зайнятим, а всі інші порожніми, або ж, якась кількість їх може бути зайнятою, а решта — ні. То що ще може поводитися як коробка? Ну, наприклад, тип `Maybe a`. У певному розумінні, це наче коробка, котра може або не містити нічого, і в такому разі вона має значення `Nothing`, або ж вона може містити лише один елемент, як-от `"HAHA"`, і в цьому випадку вона має значення `Just "HAHA"`. Ось яким чином `Maybe` є функтором.

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing  = Nothing
```

Знову ж таки, зауважте, що ми записали `instance Functor Maybe where` замість `instance Functor (Maybe m) where` так само, як ми робили, коли мали справу із `Maybe` та `YesNo`. `Functor` вимагає не конкретний тип, а конструктор типу, який приймає один тип-параметр. Якщо ви подумки замініте `f`-ки на `Maybe`, то подумки побачите, що у цьому конкретному випадку `fmap` діятиме як `(a -> b) -> Maybe a -> Maybe b`, і ця типосиґнатура є цілком «нормальною». Але якщо ви замініте `f`-ки на `(Maybe m)`, то здаватиметься, що функція `fmap` діятиме як `(a -> b) -> Maybe m a -> Maybe m b`, що не має, чорт забирай, анінайменшого сенсу, бо `Maybe` приймає лише один тип-параметр!

В усякому разі, реалізація `fmap` є доволі простою. На порожнє значення `Nothing` повертається `Nothing`. Якщо ми **відображаємо** [map over] порожню коробку, то отримуємо порожню коробку. Це має сенс. Так само, як із списками — там, якщо ми відображаємо порожній список, ми отримуємо порожній список. Якщо ж ми маємо не порожнє значення, а одне значення, «запакова-

не» у `Just`, тоді ми застосовуємо функцію до вмісту `Just`.

```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Ще одна річ, яку можна відображати і яка може втілити `Functor`, — це наш тип `Tree a`. Його теж, з натяжкою, можна уявити як коробку (містить кілька значень або жодного значення), а конструктор типу `Tree` приймає рівно один тип-параметр. Якщо ви поглянете на `fmap` так, нібито ця функція була створена лише для роботи із `Tree`, її сиґнатура типу виглядатиме як `(a -> b) -> Tree a -> Tree b`. У цьому прикладі ми скористаємося рекурсією. Порожнє дерево відображатиметься у порожнє дерево. Відображення непорожнього дерева буде деревом, яке складатиметься із (а) кореня дерева, що на вході, але із значенням, до якого було застосовано нашу функцію, і (б) лівого і правого піддерев, які будуть ті ж самі, що й в кореня дерева на вході, от лише їх буде відображено за допомогою нашої функції.

```
instance Functor Tree where
    fmap f EmptyTree                = EmptyTree
    fmap f (Node x leftsub rightsub) =
        Node (f x) (fmap f leftsub) (fmap f rightsub)
```

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3,2,1,7])
Node 28 (Node 4 EmptyTree (Node 8 EmptyTree (Node 12 EmptyTree
 (Node 20 EmptyTree EmptyTree)))) EmptyTree
```

Файно! А як щодо `Either a b`? Чи може це стати функтором? Типоклас `Functor` вимагає конструктора типу, який приймає лише один тип-параметр, але `Either` приймає аж два. Гммм... Придумав! Ми частково застосуємо `Either`, згодувавши йому один із параметрів, а другий лишивши «вільним». Ось як `Either a` є функтором у стандартній бібліотеці:

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x)  = Left x
```

Так-так, і що ж це ми тут напрограмували? Як бачимо, ми пишемо втілення для `Either a`, а не для `Either`. Це тому, що `Either a` є конструктором типу, який приймає один параметр, тоді як `Either` приймає два. Якщо б `fmap` працювала виключно із `Either a`, сиґнатура її типу була б `(b -> c) -> Either a b -> Either a c` (і це те ж саме, що й `(b -> c) -> (Either a) b -> (Either a) c`). У цій реалізації, ми відображаємо у випадку `Right`, і не відображаємо у випадку `Left`. Чому так? Що ж, якщо ми згадаємо, як було означено тип `Either a b`, то пригадається щось таке:

```
data Either a b = Left a | Right b
```

Справді, якби ми хотіли застосувати одну і ту ж функцію до них обох, `a` та `b` повинні були б мати той самий тип. Тобто, якщо б ми спробували відобразити за допомогою функції, яка приймає рядок і повертає рядок, а параметр `b` був би рядком, а параметр `a` був би числом, то це не спрацювало б. Знову ж таки, якщо уявити, що `fmap` оперує лише значеннями `Either`, то можна побачити з тієї уявної типосиґнатури, що перший параметр повинен залишатися незмінним, тоді як другий може змінюватися, і ось як раз той перший незмінний параметр і фігурує в конструкторі значень `Left`.

Це також гарно вписується в нашу **коробкову аналогію** [box analogy], якщо ми уявимо частину `Left` як порожню коробку із повідомленням про помилку, яке каже нам, чому ця коробка порожня, і яке є написаним на боці тієї коробки.

Мапи із модуля `Data.Map` теж можна зробити функторами, оскільки вони містять (чи не містять!) значення. У випадку `Map k v` функція `fmap` відобразить за допомогою функції `v -> v'` по мапі типу `Map k v` та поверне мапу типу `Map k v'`.

**Примітка**: Зауважте, що в типах `'` ніяк семантично не навантажений, так само, як і в назвах значень. Його зазвичай додають до назви, як от в `v'`, і розуміється, що `v'` — тип схожий на `v`, але трішки змінений.

Спробуйте зрозуміти самостійно, як `Map k` втілює `Functor`!

Попрацювавши із типокласом `Functor`, ми побачили, як саме типокласи можуть **представляти** [to represent] доволі круті концепції з програмування **функціями вищого порядку** [higher-order functions]. Ми також трохи попрактикувалися писати втілення і частково застосовувати конструктори типів. У одному з наступних розділів ми також розглянемо декілька законів, які справджуються для функторів.

**І ще одне!** Функтори задовольняють декільком законам, завдяки чому в них є декілька властивостей, на які ми можемо спиратися і не замислюватися забагато в роботі із ними. Якщо ми застосовуємо `fmap (+1)` до списку `[1,2,3,4]`, ми очікуємо результат `[2,3,4,5]`, а не його розвернену версію `[5,4,3,2]`. Якщо ми застосовуємо `fmap (\a -> a)` (тотожне перетворення, яке повертає просто свій єдиний параметр, без змін) до якогось списку, ми сподіваємося отримати той самий список. Наприклад, якщо б ми неправильно написали втілення функтора для нашого типу `Tree`, застосування `fmap` до дерева, у якого ліве піддерево вузла має лише елементи, які менші за значення елемента з того вузла, а праве піддерево — ті, які є більші, могло б повернути дерево, де вже немає такого впорядкування. Ми розглянемо закони функторів детальніше у одному із наступних розділів.

## 8.10 Кшталти та бойове мистецтво володіння типами

Конструктори типів приймають інші типи як параметри і врешті-решт[†] будують нам конкретні типи. Це трохи нагадує мені функції — функції приймають значення як параметри і врешті-решт повертають (будують) якісь інші значення. Ми побачили, що конструктори типів можуть бути частково застосовані (`Either String` є конструктором, який приймає один тип-параметр і повертає конкретний тип, такий як, наприклад, `Either String Int`) — і функції теж можуть. Це все є дуже і дуже цікаво. В цьому підрозділі ми більш формально розглянемо те, яким чином конструктори типів застосовуються до типів-параметрів, так само як ми формально означували (за допомогою оголошень типів), як функції застосовуються до значень.

[†]Мається на увазі, що конкретний тип отримається лише коли всі типи-параметри буде подано, а в «проміжних станах» матимемо частково застосовані конструктори типів.

**Вам не обов'язково треба продертися крізь цей підрозділ, щоб продовжити вашу магічну подорож Хаскелом**, і якщо ви його не зрозумієте, не переживайте. Однак, якщо ви таки опануєте цей розділ, це допоможе вам більш глибоко зрозуміти, як працює в Хаскелі система типів.

Отже, такі значення як `3`, `"YEAH"` та `takeWhile` (функції є також значеннями, бо ми можемо передавати їх функціям як аргументи, і таке подібне) мають кожне свій тип. Типи є маленькими ярличками, і такий ярличок є в кожного значення, і це дозволяє нам **формально міркувати** [to reason] про значення. Але типи мають свої власні маленькі ярлички, які звуться **кшталтами**. Кшталт є таким собі «типом» самого типу. Це може звучати трішки дивно і заплутано, але насправді це є крута концепція.

Що таке **кшталти** [kinds] і для чого вони годяться? Що ж, давайте перевіряти кшталти типів у GHCi за допомогою `:k`.

```
ghci> :k Int
Int :: *
```

Зірочка-сніжинка? Як незвично. Що ж це означає? `*` значить, що тип є конкретним типом. Конкретний тип є типом, який не приймає жодних типів-параметрів, а значення можуть бути лише в конкретних типів. Якщо б мені потрібно було вголос прочитати `*` (дотепер не треба було), я б сказав *зірочка* або просто *тип*.

Оукей, а зараз погляньмо, який кшталт в `Maybe`.

```
ghci> :k Maybe
Maybe :: * -> *
```

Конструктор типу `Maybe` приймає один конкретний тип (як-от `Int`) та повертає конкретний тип, такий як, наприклад, `Maybe Int`. І це все, що нам каже цей кшталт. Так само як `Int -> Int` означає, що функція приймає `Int` та повертає `Int`, кшталт `* -> *` означає, що це — конструктор типу, який приймає один конкретний тип і повертає конкретний тип. Застосуймо `Maybe` до якогось типу і гляньмо, яким є кшталт результуючого типу.

```
ghci> :k Maybe Int
Maybe Int :: *
```

Саме те, чого я й очікував! Ми подали тип-параметр до `Maybe` і отримали конкретний тип (це як раз і означає те `* -> *`). Паралеллю (хоча й не еквівалентом — типи і кшталти є різними поняттями) до цього буде виконання `:t isUpper` та `:t isUpper 'A'`. Функція `isUpper` має тип `Char -> Bool`, а `isUpper 'A'` має тип `Bool`, оскільки його значення-результат є `True`. Проте, обидва ці типи мають кшталт `*`.

Ми застосували `:k` до типу, щоб одержати його кшталт, точно так само, як ми можемо застосувати `:t` до значення, щоб отримати його тип. Як ми сказали, типи є ярличками значень, а кшталти є ярличками типів, і між ними можна провести певні паралелі.

Розгляньмо інший кшталт.

```
ghci> :k Either
Either :: * -> * -> *
```

Ага, це каже нам, що `Either` приймає два конкретні типи як типи-параметри, щоб утворити конкретний тип. Це також нагадує оголошення функції, яке приймає два значення і щось повертає. Конструктори типів є карійованими (так само, як і функції), тому ми можемо їх частково застосовувати.

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

Коли б ми хотіли зробити `Either` членом типокласу `Functor`, ми б повинні були частково застосувати його, оскільки `Functor` вимагає типи, які приймають єдиний параметр, тоді як `Either` приймає два. Іншими словами, `Functor` вимагає типи кшталту `* -> *` і таким чином нам би довелося частково застосувати `Either`, щоб отримати тип кшталту `* -> *` замість його початкового кшталту `* -> * -> *`. Якщо ми поглянемо на означення `Functor` ще раз

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

то побачимо, що змінна типу `f` використовується як тип, який приймає один конкретний тип, щоб побудувати конкретний тип. Ми знаємо, він повинен побудувати конкретний тип, бо він використовується як тип значення у функції. Звідси ми можемо зробити висновок, що типи, які хочуть товаришувати із `Functor`-ом, повинні бути кшталту `* -> *`.

А зараз ми повправляємося у бойовому мистецтві володіння типами. Розгляньмо цей типоклас, який я придумаю «на ходу»:

```
class Tofu t where
    tofu :: j a -> t a j
```

Це виглядає по-справжньому чудернацько. Якби було треба, як ми могли б створити тип, який міг би втілити такий дивний типоклас? Ну, спочатку погляньмо на те, якого кшталту він повинен був би бути. Через те, що `j a` використовується як тип значення, яке функція `tofu` приймає як параметр, `j a`

повинен мати кшталт `*`. Вважатимемо, що `a` є `*`, і з цього випливає, що `j` повинен мати кшталт `* -> *`. Бачимо далі, що конструктор `t` теж повинен збудувати нам конкретне значення і він приймає два типи. Знаючи, що `a` має кшталт `*`, а `j` має кшталт `* -> *`, ми виводимо, що `t` мусить мати кшталт `* -> (* -> *) -> *`. Таким чином, він (а) приймає (1) конкретний тип (`a`) і (2) конструктор типу (`j`), який приймає один конкретний тип, і (б) будує конкретний тип. Круто.

Добре, сотворімо тип, що має кшталт `* -> (* -> *) -> *`. Ось один спосіб, як це можна зробити.

```
data Frank a b = Frank {frankField :: b a} deriving (Show)
```

Як переконатися, що цей тип має кшталт `* -> (* -> *) - > *`? Що ж, поля в алгебраїчних типах даних можуть тримати лише значення, і тому вони, вочевидь, повинні бути типів, що мають кшталти `*`. Отож, `a` то є просто `*`, і з цього (і з означення `Frank`) випливає, що `b` приймає один параметр типу, і таким чином кшталтом `b` є `* -> *`. Тепер ми знаємо кшталти обидвох `a` та `b` і через те, що вони є параметрами конструктора типів `Frank`, ми бачимо, що цей конструктор типів має кшталт `* -> (* -> *) -> *`. Перша `*` відповідає `a`, а `(* -> *)` — `b`. Створімо декілька значень такого типу за допомогою конструктору значень `Frank`, і перевіримо, який тип має результат, на всяк випадок.

```
ghci> :t Frank {frankField = Just "HAHA"}
Frank {frankField = Just "HAHA"} :: Frank [Char] Maybe
ghci> :t Frank {frankField = Node 'a' EmptyTree EmptyTree}
Frank {frankField = Node 'a' EmptyTree EmptyTree} :: Frank Char Tree
ghci> :t Frank {frankField = "YEAH"}
Frank {frankField = "YEAH"} :: Frank Char []
```

Чому так? Оскільки `frankField` має тип `a b`, його можна ініціалізувати лише значеннями, які мають «схожий по формі» тип. Тобто, це може бути `Just "HAHA"`, яке має тип `Maybe [Char]`, або ж це може бути значення `['Y','E','S']`, яке має тип `[Char]` (якщо б ми використали наш власний списковий тип для цього, то був би тип `List Char`). І ми бачимо — типи, що їх мають значення типу `Frank`, відповідають кшталту, що його має тип `Frank`. `[Char]` має кшталт `*`, а `Maybe` має кшталт `* -> *`. Для того, щоб можна було побудувати значення якогось типу, це має бути конкретний тип, і тому, якщо маємо справу із конструктором типів, він має бути повністю застосованим. Ось чому кожне значення `Frank blah blaah` має кшталт `*`.

Втілювати `Tofu` в `Frank` доволі просто. Ми бачимо, що `tofu` приймає `j a`

(прикладом типу «такої форми» міг би бути `Maybe Int`) та повертає `t a j`. Тому, якщо ми замінимо `t` на `Frank`, тип результату (для прикладу, наведеного раніше, де `j a` є `Maybe Int`) буде `Frank Int Maybe`.

```
instance Tofu Frank where
    tofu x = Frank x
```

```
ghci> tofu (Just 'a') :: Frank Char Maybe
Frank {frankField = Just 'a'}
ghci> tofu ["HELLO"] :: Frank [Char] []
Frank {frankField = ["HELLO"]}
```

Не надто корисно, але ми принаймні розім'яли м'язи. А тепер — вйо до власне тренування! Маємо такий тип даних:

```
data Barry t k p = Barry { yabba :: p, dabba :: t k }
```

А зараз ми хочемо зробити його втіленням типокласу `Functor`. `Functor` хоче типи кшталту `* -> *`, але не схоже, що `Barry` має такий кшталт. Яким є кшталт `Barry`? Ну, бачимо, що `Barry` приймає три типи-параметри, отже, це буде `<<something>> -><<something>> -><<something>> ->*`. Особливо не напружуючись, бачимо, що `p` є конкретним типом і таким чином має кшталт `*`. Для `k` можна припустити кшталт `*`, і з цього випливає, що `t` повинен мати кшталт `* -> *`. Тепер просто замінимо ті *«something»*-заглушки на отримані кшталти, щоб отримати для `Barry` кшталт `(* -> *) -> * -> * -> *`. Перевіримо цей результат в GHCi.

```
ghci> :k Barry
Barry :: (* -> *) -> * -> * -> *
```

Ба! Ми були праві. Як приємно. Тепер, щоб зробити цей тип членом типокласу `Functor`, ми повинні частково застосувати перші два типи-параметри, щоб отримався кшталт `* -> *`. Це значить, що початком означення втілення буде `instance Functor (Barry a b) where`. Якщо ми, за звичкою, поглянемо на `fmap` так, наче вона була придумана саме для `Barry`, то ця функція матиме тип `fmap :: (a -> b) -> Barry c d a -> Barry c d b` (отримано простою заміною `f` з означення `Functor`-а на `Barry c d`. Третій тип-параметр в `Barry` буде мінятися, і як бачимо, він затишно оселився у своєму власному окремому полі.

```
instance Functor (Barry a b) where
    fmap f (Barry {yabba = x, dabba = y}) = Barry {yabba = f x, dabba = y}
```

Ось! Ми просто застосували `f` до першого поля.

Озирнемося на щойно пройдений шлях. У цьому підрозділі ми детально розглянули, як працюють параметри типів, і зробили щось на кшталт формалізації цього процесу за допомогою кшталтів, аналогічно до того, як ми формалізували параметри функцій за допомогою типосиґнатур. Ми також побачили, що можна накреслити певні паралелі між функціями та конструкторами типів. Проте вони є зовсім різні. В реальному житті вам наврядче доведеться отак ковбаситися із кшталтами і виводити кшталти вручну. Швидше за все, вам просто потрібно буде частково застосувати якийсь ваш власний тип і отримати `* -> *` чи `*`, щоб можна було втілити якийсь зі стандартних типокласів. Однак непогано знати, як і чому це насправді працює. Було цікаво дізнатися, що типи самі по собі мають свої власні маленькі «типики» — кшталти.

Знову ж таки, не обов'язково зрозуміти все, що ми щойно тут робили, аби продовжувати успішно читати решту цієї книги — але якщо ви таки зрозуміли як працюють кшталти, то скоріш за все, ви тепер добре розбираєтеся у системі типів мови Хаскел.

# Покажчик