
Про Haskell
По-людськи
Д. Шевченко

Про Haskell по-людськи

видання 2.0

Денис Шевченко
Переклав українською: Abbath
Коректор українського перекладу: Ольга Федевич

www.ohaskell.guide
2016

Зміст

Розділ 1

Вітаю!

Перед вами — книга про Haskell, дивовижну та прекрасну мову програмування.

Чому ця книга з'явилася

Тому що мене відверто дістало. Майже всі відомі мені книги про Haskell починаються з прикладу реалізації швидкого сортування і — куди ж без неї! — послідовності Фібоначчі. Ця книга не така: мінімум академізму, максимум практичності.

Мета

Функціональне програмування — своєрідне гетто посеред мегаполісу нашої індустрії. Частка функціональних мов поки що дуже мала, і багато розробників побоюються знайомства з цими мовами, і з Haskell особливо. Моя мета — зруйнувати цей страх. Ймовірно, ви

чули, що Haskell — це щось архіскладне, суто наукове і непридатне для реального життя? Читайте далі, і незабаром ви переконаєтеся в зворотньому.

Про себе

Звичайний програміст-самоучка. Розробляю з 2006 року. У 2012 році вперше почув про Haskell, жахнувся та поспішив про нього забути. У 2013 згадав знову, в 2014 захопився серйозно, а в 2015, після 8 років життя з C++, остаточно перейшов в Haskell-світ. Також я заснував російськомовне співтовариство Haskell-розробників. І так, я дійсно використовую цю мову у своїй щоденній роботі.

Про вас

Знаєте, що таке компілятор? Не боїтеся командного рядка? Чули слово «функція»? Якщо так — сміливо продовжуйте читати, ніяких додаткових навичок від вас не очікується. І якої-небудь математичної підготовки — теж.

Обіцянка

Можливо, ви по вуха закохаєтеся в Haskell. Можливо, він викличе у вас огиду. Обіцяю одне — нудно не буде. Почнемо.

Розділ 2

Перші запитання

Мені ставили їх безліч разів. Відповідаю.

«Що таке цей ваш Haskell?»

Haskell — чисто функціональна мова програмування загального призначення, може бути використана для вирішення широкого кола завдань. Компільована, але може вести себе і як скриптова. Багатоплатформенна. Ледача, з суворою статичною типізацією. І вона не схожа на інші мови. Зовсім.

«Це що, якась нова мова?»

Зовсім ні. Історія Haskell почалася ще в 1987 році. Ця мова була народжена у математичних колах, коли група людей вирішила створити найкращу функціональну мову програмування. У 1990 році

вийшла перша версія мови, названої на честь відомого американського математика Хаскела Каррі. У 1998 році мова була стандартизована, а починаючи з 2000-х почалося її повільне входження у світ практичного програмування. За ці роки мова удосконалювалася, і ось в 2010 побачив світ її оновлений стандарт. Так що ми маємо справу з мовою, яка старша ніж Java.

«І хто його зробив?»

Haskell створювався багатьма людьми. Найбільш відома реалізація мови знайшла своє втілення в компіляторі GHC (The Glasgow Haskell Compiler), який народився в 1989 році в Університеті Глазго. У компілятора було кілька головних розробників, з яких найбільш відомі двоє, Simon Peyton Jones та Simon Marlow. Згодом вагомий внесок у розробку GHC внесли ще кілька сотень людей. Вихідний код компілятора GHC відкритий. До речі, сам компілятор на 82% написаний на Haskell.

Для цікавих: вичерпну розповідь про історію Haskell і GHC читайте тут.

«А бібліотеки для Haskell є?»

О так! Їх навіть не сотні, їх тисячі. У процесі читання ви познайомитеся з багатьма із них.

«І що, його вже можна в production?»

Він вже в production. З моменту виходу першого стандарту мова поліпшувалася, розвивалася її екосистема, з'являлися нові бібліотеки, виходили у світ книги. Сьогодні, в 2016, можна впевнено заявити, що Haskell повністю готовий до серйозного комерційного використання, про що свідчать історії успішного впровадження Haskell в бізнесі, в тому числі великому.

«А поріг входження в Haskell високий?»

І так, і ні. Освоєння Haskell складне в першу чергу через його несхожість на інші мови, тому людям, що мають досвід роботи з іншими мовами, мізки поламати доведеться. Саме поламати, а не просто поворушити ними: Haskell змушує інакше поглянути навіть на звичні речі. З іншого боку, Haskell простіший від багатьох відомих мов. Не вірте мені на слово, незабаром ви й самі в цьому переконаєтеся. І знайте: багато людей, дізнавшись смак Haskell, категорично не бажають повертатися до інших мов. Я вас попередив.

«А я чув ще про якісь монади...»

Так, є така справа. Деякі речі з світу Haskell не мають прямих аналогів в інших мовах програмування, і це вводить новачків в ступор. Але не турбуйтеся: я сам пройшов через цей ступор і добре вас розумію. Пам'ятайте: нове лише здається страшним.

«А якщо порівняти його з C++/Python/Scala…»

Порівняння Haskell з іншими мовами виходить за рамки цієї книги. Кілька разів ви зустрінете тут шматочки коду на інших мовах, але я наводжу їх виключно для того, щоб підкреслити відмінність з Haskell, а зовсім не для порівняння в контексті «краще/гірше».

Розділ 3

Про цю книгу

В останні роки помітно зросла кількість книг, присвячених Haskell, і це тішить. Кожна з них переслідує свою мету, тому важко сказати, яка з них краща. Мета цієї книги двояка.

По-перше, я навчу вас головному в Haskell. Основ, без освоєння яких рухатися далі ніяк не вийде.

По-друге, я зруйную страх. Вже багато років довкола Haskell витає дух страху, і я сповна відчув його на собі. Насправді Haskell зовсім не страшний, в ньому немає чорної магії, і щоб програмувати на ньому, вам не потрібен вчений ступінь. Більш того, ви здивуєтеся, наскільки просто в Haskell робити багато речей, але ця простота відкриється вам лише після того, як ви близько познайомитеся з Трьома Китами Haskell, а також з пані Черепахою, що підтримує їх. Імена цих Китів і Черепахи ви дізнаєтеся вже в наступній главі.

Ця книга не заведе вас на вершини Haskell, але вона відкриє вам шлях до цих вершин.

Чого тут немає

Трьох речей, які ви не знайдете на сторінках цієї книги:

1. Вичерпного довідника по Haskell. Дублювати офіційний опис стандарту Haskell 2010 я не стану.
2. Набору готових рецептів. За рецептами завітайте на Stackoverflow.
3. Введення в математичну теорію. Незважаючи на те, що Haskell корінням своїм йде в математику, у цій книзі немає занурення в теорію категорій та інші теорії. Вибачте, якщо розчарував.

Про перше та друге видання

На обкладинці ви бачили мітку «видання 2.0». Перед вами друге видання, повністю перероблене та переосмислене. Ось дві причини, що спонукали мене переписати книгу.

Перша — мої помилки. Я переконаний, що навчати мови програмування можуть лише ті, хто використовує цю мову у своїй щоденній роботі. На момент написання першої версії я ще не працював з Haskell, а тому багато чого не знав і не розумів. В результаті частина інформації з першого видання була відверто бідною, а кілька глав взагалі вводили читача в оману.

Друга причина — змінилася мета книги. Я навмисно звузив коло розглянутих тут тем. Тепер книга цілком присвячена основам мови, тому не шукайте тут розгляду специфічних тем. Я не дуже-то вірю в ідею book-all-in-one, книга для новачків повинна бути книгою для новачків. Ви не зустрінете тут ні прикладів реалізації 3D-рушія, ні розповіді про роботу з PostgreSQL, ні розповіді про проектування гри для Android. Все це можна робити з Haskell, але подібним темам присвячені інші публікації, які безсумнівно будуть вам до снаги після прочитання моєї книги.

Читайте послідовно

І це важливо. У процесі читання ви помітите, що я періодично піднімаю питання і неначе залишаю їх без відповіді. Це робиться цілком свідомо: відповіді обов'язково будуть дані, але в наступних розділах, там, де це буде найбільш доречно. Тому перестрибування з розділу на розділ може вас заплутати.

Втім, в веб-версії книги є «Предметний покажчик», який допоможе вам швидко знайти потрібне місце, що особливо корисно при повторному прочитанні книги.

Для цікавих

Наприкінці більшості розділів ви знайдете невелику секцію, яка так і називається — «Для допитливих». Читати її не обов'язково, але допитливим неодмінно сподобається. У цій секції я наводжу деякі технічні подробиці, історичні відомості та просто цікаві факти.

І врахуйте, будь ласка: вміст секції «Для допитливих» іноді трохи ламає послідовність викладу матеріалу, це зроблено свідомо. Пам'ятаючи про численні питання читачів до розділів з попереднього видання, я виніс відповіді на деякі з цих питань в дану секцію, і тому вона, скажімо, в 12 розділі може посилатися на матеріал, викладений лише в 16 розділі. Якщо сумніваєтеся — не читайте.

Про пояснення

У багатьох прикладах вихідного коду ви побачите пояснення ось такого вигляду:

```
type String =     [Char]

тип  цей дорівнює цьому
```

Такі пояснення слід читати зліва направо і зверху вниз, і ви відразу зрозумієте що до чого. Кожна частина пояснення розташована строго під тим шматочком коду, до якого вона належить.

Ось ще один приклад:

```
let (host, alias) = ("173.194.71.106", "www.google.com")

                        дане значення
     це
     хост
                                          а ось це
                                          значення
          це
          ім'я
```

Тут я кажу вам: «Дане значення — це хост, а ось це значення — це ім'я». У ряді випадків я використовую також різного виду підкреслення:

```
(host, alias) = ("173.194.71.106", "www.google.com")

____           ________________

       =====                     ================
```

Тут я проводжу паралель: «Значення `host` асоційоване з рядком `173.194.71.106`, а значення `alias` — з рядком `www.google.com`».

Подяка

Ця книга — плід не тільки моїх зусиль. Багато членів нашої спільноти допомогли мені порадами, зауваженнями та виправленнями. Велике спасибі вам, друзі!

А ще я дякую всім тим, хто створив Haskell, і всім тим, хто невпинно вдосконалює його. З вашими зусиллями наша професія стає ще більш прекрасною!

Слово до тих, хто читав перше видання

Якщо ви не читали його — можете переходити до наступного розділу.

Як вже було сказано, мета книги змінилася. Я переконаний, що новачкові слід дати фундамент, освоївши який, він зможе самостійно вивчати те, що потрібно саме йому. Я більше не хочу давати читачам рибу, я хочу дати їм вудку. Тому тут немає розповідей про всі наявні монадні трансформери, або про всі контейнери, або про Кметтівські лінзи, або про труби Гонзалеса.

Я зроблю акцент на теорію, але вже глибше. Так, в минулому виданні я часто використовував неточну термінологію, відверто затупив з визначенням монади, прогнав якусь пургу з ФВП, ні словом не обмовився про функторні та інші закони, майже не розповів про патерн матчінг і використовував мало прикладів реального коду. У цьому виданні я постараюся виправити ці помилки.

І я як і раніше відкритий до вашої критики.

Розділ 4

Приготуймося

Ми не можемо почати вивчення мови без випробувального полігону. Встановимо Haskell.

Зробити це можна кількома способами, ми виберемо найбільш зручний. Називається він The Haskell Tool Stack. Ця маленька утиліта — все, що вам знадобиться для роботи з Haskell.

Haskell — багатоплатформенна мова, працює і в OS X і Linux, і навіть в Windows. Однак у 2008 році я назавжди покинув світ Windows, тому всі наступні приклади взаємодії з командним рядком мають на увазі Unix-way. Строго кажучи, я вже й забув, що таке командний рядок в Windows.

Вся конфігурація та приклади коду випробувані мною на OS X Yosemite.

Встановлюємо

Йдемо сюди і завантажуємо архів для потрібної нам ОС. Розпаковуємо архів і бачимо програмку під назвою `stack`. Для зручності збе-

рігаємо її в якомусь каталозі, доступному в PATH. Рекомендований шлях — ~/.local/bin/.

Якщо ж ви живете в світі Mac і користуєтеся Homebrew, вам ще простіше. Робите:

```
$ brew update
$ brew install haskell-stack
```

Все.

На момент написання книги я використовував stack версії 1.0.2. Якщо у вас стара версія — неодмінно оновіть систему. Якщо ж більш нова — у вас теоретично щось може працювати не точно так, як описано нижче, оскільки stack активно розвивається.

Головне (але не єдине), що вміє робити stack, це:

1. Розгортати інфраструктуру.
2. Збирати проекти.
3. Встановлювати бібліотеки.

Haskell-інфраструктура — екосистема, наріжним каменем якої є раніше згаданий компілятор GHC. Haskell є компільованою мовою: додаток являє собою звичайний виконуваний (англ. executable) файл.

Haskell-проект — середовище для створення додатків і бібліотек.

Haskell-бібліотеки — кимось написані рішення, які рятують нас від винаходу велосипеда.

Розгортаємо інфраструктуру

Робимо:

```
$ stack setup
```

В результаті на ваш комп'ютер буде встановлена інфраструктура останньої стабільної версії. Жити все це господарство буде у тільки що створеному каталозі `~/.stack/`. Саме тому встановлювати інфраструктуру для наступних Haskell-проектів вам вже не доведеться: одноразово розгорнули, використовуємо завжди. Поки вам не потрібно знати про устрій цієї інфраструктури, сприймайте її як дане: тепер на вашому комп'ютері живе Haskell.

Hi World

Створимо наш перший Haskell-проект:

```
$ stack new real
```

Тут `real` — назва проекту. В результаті буде створено каталог `real`, всередині якого ми побачимо це:

```
.
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs <- Головний модуль
├── real.cabal <- Збірочний файл проекту
├── src
│   └── Lib.hs <- Допоміжний модуль
├── stack.yaml
└── test
    └── Spec.hs
```

Про вміст проекту вам знати не потрібно, просто зберемо його командою:

```
$ stack install
```

Запам'ятайте цю команду, ми будемо використовувати її постійно. У результаті з'явиться файл `real-exe`. А оскільки він буде скопійований у згаданий вище каталог `~/.local/bin/`, ми зможемо відразу запустити програму:

```
$ real-exe
someFunc
```

Ось ми і створили Haskell-проект і запустили нашу першу програму, що вивела рядок `someFunc`. Але як же це працює? Прийшла пора познайомитися з фундаментальною одиницею проекту — модулем.

Модулі: знайомство

Haskell-проект складається з модулів. Модулем називається файл, який містить вихідний Haskell-код. Один файл — один модуль. Розширення `.hs` — стандартне розширення для модулів. В Haskell немає поняття «файлу заголовку»: кожен з модулів розглядається як самостійна одиниця проекту, що містить у собі різні корисні речі. А щоб скористатися цими речами, необхідно один модуль імпортувати в інший.

Відкриємо модуль `src/Lib.hs`:

```haskell
module Lib -- Ім'я модуля
( someFunc
) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

У першому рядку оголошено, що ім'я цього модуля — `Lib`. Далі у круглих дужках згадано вміст даного модуля, а саме ім'я функції `someFunc`. Потім, після ключового слова `where`, ми бачимо визначення функції `someFunc`. Поки вам не потрібно знати про синтаксис цієї конструкції, у наступних розділах ми розберемо його детальніше.

Тепер відкриємо модуль `app/Main.hs`:

```haskell
module Main where

import Lib -- Імпортуємо модуль Lib...

main :: IO ()
main = someFunc -- Використовуємо його вміст...
```

Це модуль `Main`, головний модуль нашого додатка, адже саме тут визначена функція `main`. За допомогою директиви `import` ми включаємо сюди модуль `Lib` і можемо працювати з вмістом цього модуля.

Запам'ятайте модуль `Main`, з ним ми будемо працювати найчастіше. Всі приклади вихідного коду, які ви побачите на сторінках цієї книги, проживають саме в модулі `Main`, якщо не обумовлено інше.

Всі модулі в наших проектах можна розділити на дві частини: ті, які ми беремо з бібліотек і ті, які ми створили самі. Бібліотеки — це вже кимось написані рішення, в наступних розділах ми познайо-

мимося з багатьма із них. Серед бібліотек слід виділити одну, так звану стандартну бібліотеку. Модулі із стандартної бібліотеки ми почнемо використовувати вже в наступних розділах. А один з розділів буде повністю присвячений розповіді про бібліотеки: з нього ми детально дізнаємося, звідки беруться бібліотеки та як їх можна використовувати.

Для допитливих

До появи `stack` основним способом установки Haskell була так звана Haskell Platform. Однак саме `stack`, незважаючи на свою молодість (вийшов у світ влітку 2015 року), є найкращим шляхом у світ Haskell, особливо для новачків. Якщо раптом так сталося, що до моменту прочитання цього розділу у вас вже була встановлена Haskell Platform — конче рекомендую вам негайно видалити цей мотлох. Якщо у вас OS X, ви можете скористатися ось цим маленьким скриптом.

Як ви помітили, імена файлів з вихідним кодом починаються з великої букви: `app/Main.hs` і `src/Lib.hs`. Строго кажучи, це необов'язково, можна і з маленької букви, однак для гармонії з іменем модуля краще дотримуватися загальноприйнятої практики і називати файл модуля за іменем самого модуля:

```
app/Main.hs -> module Main ...
src/Lib.hs -> module Lib ...
```

Розділ 5

Кити і Черепаха

Отже, проект створили, тепер ми готові розпочати нашу подорож.

Haskell стоїть на Трьох Китах, імена яких: **Функція**, **Тип** і **Клас типів**. Вони ж, у свою чергу, спочивають на величезній Черепасі, ім'я якої **Вираз**.

Черепаха

Haskell-програма є сукупністю виразів (англ. expression). Погляньте:

```
1 + 2
```

Це — основна цеглина Haskell-програми, було б це Hello World або частина інфраструктури міжнародного банку. Звісно, крім додавання одиниці з двійкою існують й інші вирази, але суть в них у всіх одна:

> Вираз — це те, що може дати нам певний корисний результат.

Всі вирази можна розділити на дві групи: ті, які (все ще) можна обчислити й ті, що (вже) не можна. Обчислення (англ. evaluation) — це фундаментальна дія по відношенню до виразу, адже саме обчислення дає нам той самий корисний результат. Так, вираз:

```
1 + 2
```

може дати нам корисний результат, а саме суму двох чисел. Обчисливши вираз, ми отримуємо результат:

```
3
```

Причому це не просто число `3`, це теж вираз. Подібний вираз вже не можна обчислити, він обчислений остаточно, до самого дна, і ми можемо лише використовувати його як є.

В результаті обчислення вираз завжди зменшується (англ. reduce). В україномовній літературі іноді так і пишуть: «редукція виразів». Зменшувати вираз можна до тих пір, поки він не досягне своєї нередукованої форми. Згаданий вище вираз `1 + 2` ще можна редукувати, а от вираз `3` — вже не можна.

Таким чином, вирази, складові програми, обчислюються/редукуються до тих пір, поки не залишиться якийсь остаточний, кореневий вираз. А запуск Haskell-програми на виконання (англ. execution) — це запуск всього цього ланцюжка обчислень, причому з коренем цього ланцюжка ми вже познайомилися раніше. Пам'ятаєте функцію `main`, визначену в модулі `app/Main.hs`? Ось ця функція і є головною точкою нашої програми, її Альфою та Омегою.

Перший Кит

Повернемося до виразу `1 + 2`. Корисний результат ми отримаємо лише після того, як обчислимо вираз, тобто здійснимо додавання. І як же можна «здійснити додавання» в рамках Haskell-програми? За допомогою функції. Саме функція робить вираз обчислюваним, саме вона оживляє нашу програму, тому я й назвав Функцію Першим Китом Haskell. Але щоб уникнути непорозумінь, визначимося з поняттями.

Згадаймо математичне визначення функції. Не лякайтеся, математики буде зовсім небагато:

> Функція — це закон, що описує залежність одного значення від іншого.

Розглянемо функцію піднесення цілого числа до квадрату:

```
square v = v * v
```

Функція `square` визначає просту залежність: числу `2` відповідає число `4`, `3` — `9`, і так далі. Схематично це можна записати так:

```
2 -> 4
3 -> 9
4 -> 16
5 -> 25
...
```

Вхідне значення функції називають аргументом. А оскільки функція визначає однозначну залежність вихідного значення від аргументу, її, функцію, називають ще *відображенням*: вона відображає/проектує вхідне значення на вихідне. Виходить неначе труба: кинули в неї `2` — `3` іншого боку вилетіло `4`, кинули `5` — вилетіло `25`.

Щоб змусити функцію зробити корисну роботу, її необхідно застосувати (англ. apply) до аргументу. Приклад:

```
square 2
```

Ми застосували функцію `square` до аргументу `2`. Синтаксис гранично простий: ім'я функції та через пробіл аргумент. Якщо аргументів більше одного — просто дописуємо їх так само, через пробіл. Наприклад, функція `sum`, яка обчислює суму двох своїх цілочисельних аргументів, застосовується так:

```
sum 10 20
```

Так от вираз `1 + 2` є ніщо інше, як застосування функції! І щоб ясніше побачити, перепишемо вираз:

```
(+) 1 2
```

Це застосування функції `(+)` до двох аргументів, `1` і `2`. Не дивуйтеся, що ім'я функції вставлено в дужки, незабаром я розповім про це детальніше. А поки запам'ятайте головне:

> Обчислити вираз — це значить застосувати якісь функції (одну або більше) до якихось аргументів (одного або більше).

І ще. Можливо, ви чули про так званий «виклик» функції. В Haskell функції не викликають. Поняття «виклик» функції прийшло до нас з поважної мови C. Там функції дійсно викликають (англ. call), тому що в C, на відміну від Haskell, поняття «функція» не має ніякого відношення до математики. Там це підпрограма, тобто відокремлений шматочок програми, доступний за певною адресою в пам'яті. Якщо у вас є досвід розробки на C-подібних мовах — забудьте про підпрограми. В Haskell функція це функція в математичному сенсі

слова, тому її не викликають, а застосовують до чогось.

Другий Кит

Отже, будь-який редукований вираз це застосування функції до деякого аргументу (теж є виразом):

```
square  2
функція аргумент
```

Аргумент являє собою деяке значення, його ще називають «даними» (англ. data). Дані в Haskell — це сутності, які володіють двома головними характеристиками: типом і конкретним змістом/вмістом.

Тип — це Другий Кит в Haskell. Тип відображає конкретний вміст даних, а тому всі дані в програмі обов'язково мають якийсь тип. Коли ми бачимо дані типу `Double`, ми точно знаємо, що перед нами число з плаваючою крапкою, а коли бачимо дані `String` — можемо бути певні, що перед нами рядок.

Ставлення до типів в Haskell дуже серйозне, і робота з типами характеризується трьома важливими рисами:

1. Статична перевірка,
2. Сила,
3. Виведення.

Ці три властивості системи типів Haskell — наші добрі друзі, адже вони роблять наше програмістське життя щасливішим. Познайомимося з ними.

Статична перевірка

Статична перевірка типів (англ. static type checking) — це перевірка всіх типів даних в програмі, яка здійснюється на етапі компіляції. Haskell-компілятор впертий: коли йому щось не подобається в типах, він голосно лається. Тому якщо функція працює з цілими числами, застосувати її до рядків ніяк не вийде. Так що якщо компіляція нашої програми завершилася успішно, ми точно знаємо, що з типами у нас все в порядку. Переваги статичної перевірки неможливо переоцінити, адже вона гарантує відсутність в наших програмах цілого ряду помилок. Ми вже не зможемо сплутати числа з рядками або відняти метри від гривень.

Звичайно, у цієї медалі є і зворотня сторона час, який витрачається на компіляцію. Вам доведеться звикнутися з цією думкою: внесли зміни в проект — будьте ласкаві скомпілювати. Однак втіхою вам нехай послужить той факт, що переваги статичної перевірки куди цінніші за час, витрачений на компіляцію.

Сила

Сильна (англ. strong) система типів — це безкомпромісний контроль відповідності очікуваного дійсному. Сила робить роботу з типами ще більш акуратною. Ось вам приклад зі світу C:

```
double coeff(double base) {
return base * 4.9856;
}

int main() {
int value = coeff(122.04);
...
}
```

Це канонічний приклад проблеми, обумовленої слабкою (англ. weak) системою типів. Функція `coeff` повертає значення типу `double`, однак викликаюча сторона очікує чомусь ціле число. Ну ось ми помилилися, криво скопіювали. В цьому випадку відбудеться шахрайство, назване прихованим приведенням типів (англ. type casting): число з плаваючою крапкою, повернуте функцією `coeff`, буде брутально зламано шляхом приведення до типу `int`, в результаті чого дробова частина буде відкинута і ми отримаємо не `608.4426`, а `608`. Подібна помилка, до речі, призводила до серйозних наслідків, таких як руйнування космічних апаратів.

В Haskell подібний код не має жодних шансів пройти компіляцію. Ми завжди отримуємо те, що очікуємо, і якщо має бути число з плаваючою точкою — хоч убийся, але надай саме його. Компілятор скрупульозно відстежує відповідність очікуваного типу фактичному, тому коли компіляція завершується успішно, ми абсолютно впевнені в гармонії між типами всіх наших даних.

Виведення

Виведення (англ. inference) типів — це здатність визначити тип даних автоматично з конкретного виразу. У тій же мові C тип даних слід вказувати явно:

```
double value = 122.04;
```

проте в Haskell ми напишемо просто:

```
value = 122.04
```

У цьому випадку компілятор автоматично виведе тип `value` як `Double`.

Виведення типів робить наш код лаконічнішим і простішим в супроводі. Втім, ми можемо вказати тип значення і явно, а іноді навіть повинні це зробити. У наступних розділах я поясню, чому.

Так, до речі, ось найпростіші стандартні типи, вони нам знадобляться:

```
123 Int
23.5798 Double
'a' Char
"Hello!" String
True Bool, істина
False Bool, хиба
```

З типами `Int` і `Double` ви вже знайомі. Тип `Char` — це Unicode-символ. Тип `String` — звичайний рядок. Тип `Bool` — логічний тип, що відповідає істині або хибі. У наступних розділах ми зустрінемося ще з кількома стандартними типами, але поки вистачить і цих. Зауважте: ім'я типу в Haskell завжди починається з великої літери.

Третій Кит

А ось про Третього Кита, про **Класи типів**, я поки промовчу, тому що знайомитися з ним слід лише після того, як ми ближче потоваришуємо з першими двома.

Впевнений, після прочитання цього розділу, з'явилося безліч запитань. Відповіді будуть, але пізніше. Більше того, наступний розділ безсумнівно здивує вас.

Для допитливих

Якщо ви працювали з об'єктно-орієнтованими мовами, такими як C++, вас здивує той факт, що в Haskell між поняттями «тип» і «клас» проведено чітку межу. А оскільки типам і класам типів в Haskell відведена колосально важлива роль, добра порада для вас: коли у майбутніх розділах ми познайомимося з ними ближче, не намагайтеся проводити аналогії з іншими мовами. Наприклад, деякі вбачають спорідненість між класами типів в Haskell й інтерфейсами в Java. Не робіть цього, щоб уникнути плутанини.

Розділ 6

Незмінність і чистота

У попередньому розділі ми ознайомилися з функціями та виразами, побачили близький зв'язок цих понять. У цьому розділі ми познайомимося з функціями ближче, а також дізнаємося, що таке «чисто функціональна» мова та чому в ній немає місця для оператора присвоювання.

Оголошуємо та визначаємо

Застосування функції нам вже знайоме, залишилося дізнатися про оголошення та визначення, без них використовувати функцію не вийде. Пам'ятаєте функцію `square`, яка підносить свій єдиний аргумент до квадрату? Ось як виглядає її оголошення та визначення:

```
square :: Int -> Int
square v = v * v
```

Перший рядок містить оголошення, другий — визначення. Оголошення (англ. declaration) — це повідомлення всьому світу про те, що така функція існує, ось її ім'я, а ось типи, з якими вона працює. Визначення (англ. definition) — це звістка про те, що конкретно робить дана функція.

Розглянемо оголошення:

```
square :: Int -> Int
```

Воно розділене подвійною двокрапкою на дві частини: зліва вказано ім'я функції, праворуч — типи, з якими ця функція працює, а саме типи аргументів і тип обчисленого, підсумкового значення. Як ви дізналися з попереднього розділу, всі дані в Haskell-програмі мають конкретний тип, а оскільки функція працює з даними, її оголошення містить типи цих даних. Типи розділені стрілками. Схематично це виглядає так:

```
square :: Int   ->   Int

ім'я       тип         тип
функції    аргументу   обчисленого
                       значення
```

Таке оголошення каже нам про те, що функція `square` приймає єдиний аргумент типу `Int` і повертає значення того ж типу `Int`. Якщо ж аргументів більше одного, оголошення просто витягується. Наприклад, оголошення функції `prod`, яка повертає добуток двох цілих аргументів, могло б виглядати так:

```
prod ::  Int  ->   Int   ->  Int

ім'я     тип       тип       тип
функції  першого   другого   обчисленого
         аргументу аргументу значення
```

Ідею ви зрозуміли: шукаємо крайню праву стрілку, і все, що знаходиться лівіше від неї — це типи аргументів, а все, що знаходиться правіше — це тип обчисленого значення.

Ми не можемо працювати з функцією, яка нічого не обчислює. Тобто аналога С-функції `void f(int i)` в Haskell бути не може, так як це суперечить математичній природі. Однак ми можемо працювати з функцією, яка нічого не приймає, тобто з аналогом С-функції `int f(void)`. З такими функціями ми познайомимося в наступних розділах.

Тепер розглянемо визначення функції `square`:

```
square v = v * v
```

Схема визначення така:

```
square   v =       v * v

ім'я     ім'я      це вираз
функції  аргументу
```

А функція `prod` визначена так:

```
prod     x         y    =   x * y

ім'я     ім'я      ім'я     це вираз
функції  першого   другого
         аргументу аргумент
```

Визначення теж поділено на дві частини: ліворуч від знака рівності — ім'я функції і список аргументів (імена, а не типи), розділені пробілами, а праворуч — вираз, що становить суть функції, її вміст. Іноді ці частини називають «головою» і «тілом»:

```
square v       = v * v

голова функції   тіло функції
(англ. head)     (англ. body)
```

Зверніть увагу, мова тут йде саме про знак рівності, але в жодному разі не про оператор присвоювання. Ми нічого не присвоюємо, ми лише декларуємо рівність лівої та правої частин. Коли ми пишемо:

```
prod x y = x * y
```

ми оголошуємо наступне: «Відтепер вираз `prod x y` тотожний виразу `x * y`». Ми можемо безпечно замінити вираз `prod 2 5` виразом `2 * 5`, а вираз `prod 120 500` — виразом `120 * 500`, і при цьому робота програми гарантовано залишиться незмінною.

Але звідки у мене така впевненість? А ось звідки.

Чисто функціональна

Haskell — чисто функціональна (англ. purely functional) мова. Чисто функціональним вона називається тому, що центральне місце в ній приділено чистій функції (англ. pure function). А чистою називається така функція, яка гранично чесна з нами: її вихідне значення цілком визначається її аргументами і більше нічим. Це і є функція в математичному сенсі. Згадаймо функцію `prod`: коли на вході числа `10` і `20` — на виході завжди буде `200`, і ніщо не здатне перешкодити цьому. Функція `prod` є чистою, а тому характеризується відсутністю сторонніх ефектів (англ. side effects): вона не здатна зробити нічого, крім повернення добутку двох своїх аргументів. Саме тому чиста функція гранично надійна, адже вона не може зробити нам ніяких сюрпризів.

Скажу більше: чисті функції не бачать навколишній світ. Взагалі. Вони не можуть вивести текст на консоль, їх не можна змусити обробити HTTP-запит, вони не вміють дружити з базою даних і прочитати файл вони нездатні. Вони є річчю в собі.

А щоб здивувати вас ще більше, відкрию ще один секрет Haskell.

«Присвоювання? Ні, не чув...»

У світі Haskell немає місця для оператора присвоювання. Втім, цей факт дивний лише на перший погляд. Замислимося: якщо кожна функція в кінцевому підсумку є виразом, обчислюваним за допомогою застосування якихось інших функцій до якихось інших аргументів, тоді нам просто не потрібно нічого нічому присвоювати.

Згадаймо, що присвоювання (англ. assignment) прийшло до нас з імперативних мов. Імперативне програмування (англ. imperative

programming) — це напрямок у розробці, що об'єднує кілька парадигм програмування, однією з яких є знаменита об'єктно-орієнтована парадигма. В рамках цього напряму програма сприймається як набір інструкцій, виконання яких нерозривно пов'язане зі зміною стану (англ. state) цієї програми. Ось чому в імперативних мовах обов'язково присутнє поняття «змінної» (англ. variable). А раз є змінні — повинен бути оператор присвоювання. Коли ми пишемо:

```
coeff = 0.569;
```

ми тим самим наказуємо: «Візьми значення `0.569` і перезапиши його поверх того значення, яке вже містилося в змінній `coeff` до цього». І перезаписувати це значення ми можемо багато разів, а отже, ми змушені уважно відслідковувати поточний стан змінної `coeff`, так само як і стан всіх інших змінних в нашому коді.

Однак існує принципово інший підхід до розробки, а саме декларативне програмування (англ. declarative programming). Даний напрямок також включає в себе кілька парадигм, однією з яких є функціональна парадигма, що знайшла своє втілення в Haskell. У цьому підході програма сприймається вже не як набір інструкцій, а як набір виразів. А оскільки вирази обчислюються шляхом застосування функцій до аргументів (тобто, по суті, до інших виразів), там немає місця ні для змінних, ні для оператора присвоювання. Всі дані в Haskell-програмі, будучи створеними один раз, вже не можуть бути змінені. Тому нам не потрібен не тільки оператор присвоювання, але і ключове слово `const`. І коли в Haskell-коді ми пишемо:

```
coeff = 0.569
```

ми просто оголошуємо: «Відтепер значення `coeff` дорівнює `0.569`, і так воно буде завжди». Ось чому в Haskell-коді символ `=` — це знак рівності в математичному сенсі, і з присвоюванням він не має ні-

чого спільного.

Упевнений, ви здивовані. Як же можна написати реальну програму на мові, в якій можна редагувати дані? Який зиск від цих чистих функцій, якщо вони не здатні ні файл прочитати, ні запит по мережі відправити? Виявляється, користь є, й на Haskell можна написати дуже навіть реальну програму. За прикладом далеко ходити не буду: сама ця книга побудована за допомогою програми, написаної на Haskell, про що я докладніше розповім в наступних главах.

А тепер, щоб не мучити вас питаннями без відповіді, ми почнемо ближче знайомитися з Китами Haskell, і деталі великої головоломки поступово складуться в красиву картину.

Для допитливих

У процесі роботи Haskell-програми в пам'яті створюється безліч різних даних, адже ми постійно будуємо нові дані на основі вже наявних. За їх своєчасне знищення відповідає збирач сміття (англ. garbage collector, GC), вбудовуваний в програми компілятором GHC.

Розділ 7

Вибираємо та повертаємося

У цій главі ми зустрінемося з умовними конструкціями, зазирнемо в термінал, а також дізнаємося, чому з Haskell-функцій не повертаються (втім, останнє — не більше ніж гра слів).

Зазирнемо у зовнішній світ

Ми починаємо писати справжній код. А для цього нам знадобиться вікно у зовнішній світ. Відкриємо модуль `app/Main.hs`, знайдемо функцію `main` і напишемо в ній наступне:

```
main :: IO ()
main = putStrLn "Hi, real world!"
```

Стандартна функція `putStrLn` виводить рядок на консоль. А якщо говорити строгіше, функція `putStrLn` застосовується до значення типу

`String` і робить так, щоб ми побачили це значення в нашому терміналі.

Так, я вже чую питання уважного читача. Як же так, запитаєте ви, хіба ми не говорили про чисті функції в минулому розділі, які не можуть взаємодіяти з зовнішнім світом? Доведеться зізнатися: функція `putStrLn` відноситься до особливих функцій, які можуть-таки вилізти в зовнішній світ. Але про це в наступних розділах. Це дуже цікава тема, повірте мені!

І ще нам слід познайомитися з Haskell-коментарями, вони нам знадобляться:

```
{-
  Я - складний багаторядковий
  коментар, що містить
  щось
  дуже важливе!
-}
main :: IO ()
main =
  -- А я - скромний однорядковий коментар.
  putStrLn "Hi, real world!"
```

Символи `{-` і `-}` приховують багаторядковий коментар, а символ `--` починає однорядковий коментар.

Про всяк випадок нагадую команду збірки, запускається з кореня проекту:

```
$ stack install
```

Після збірки запускаємо проект:

```
$ real-exe
Hi, real world!
```

Вибір і вихід

Вибирати всередині функції доводиться дуже часто. Існує декілька способів задання умовної конструкції. Ось базовий варіант:

```
if CONDITION then EXPR1 else EXPR2
```

де `CONDITION` — логічний вираз, що дає хибу або істину, `EXPR1` — вираз, що використовується в разі `True`, `EXPR2` — вираз, що використовується в разі `False`. Приклад:

```
checkLocalhost :: String -> String
checkLocalhost ip =
  -- True або False?
  if ip == "127.0.0.1" || ip == "0.0.0.0"
    -- Якщо True - йде туди...
    then "it's a localhost!"
    -- А якщо False - сюди...
    else "No, it's not a localhost."
```

Функція `checkLocalhost` застосовується до єдиного аргументу типу `String` і повертає інше значення типу `String`. Аргументом виступає рядок, що містить IP-адресу, а функція перевіряє, чи не лежить в ній localhost. Оператор `||` — стандартний оператор логічного «АБО», а оператор `==` — стандартний оператор перевірки на рівність. Отже, якщо рядок `ip` дорівнює `127.0.0.1` або `0.0.0.0`, значить в ньому міститься localhost, і ми повертаємо перший вираз, тобто рядок `it's a localhost!`, в іншому випадку повертаємо другий вираз, рядок `No, it's not a localhost.`.

А до речі, що означає «повертаємо»? Адже, як ми дізналися, функції в Haskell не викликають (англ. call), а значить, з них і не повертаються (англ. return). І це дійсно так. Якщо напишемо:

```
main :: IO ()
main = putStrLn (checkLocalhost "127.0.0.1")
```

при запуску побачимо це:

```
It's a localhost!
```

а якщо так:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")
```

тоді побачимо це:

```
No, it's not a localhost.
```

Круглі дужки включають вираз типу `String` за схемою:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")

                |___   вираз типу String   ___|
```

Тобто функція `putStrLn` бачить не застосування функції `checkLocalhost` до рядка, а просто вираз типу `String`. Якщо б ми опустили дужки і написали так:

```
main :: IO ()
main = putStrLn checkLocalhost "173.194.22.100"
```

сталася помилка компіляції, і це цілком очікувано: функція `putStrLn` застосовується до одного аргументу, а тут їх виходить два:

```
main = putStrLn        checkLocalhost ”173.194.22.100”

       функція         до цього
       застосовується  аргументу...
                                      і до цього??
```

Не знаю як ви, а я не дуже люблю круглі дужки, при всій повазі до Lisp-програмістів. На щастя, в Haskell існує спосіб зменшити кількість дужок. Про цей спосіб — в одному з наступних розділів.

Так що ж з поверненням з функції? Згадаймо про рівність у визначенні:

```
checkLocalhost ip =
  if ip == ”127.0.0.1” || ip == ”0.0.0.0”
    then ”it’s a localhost!”
    else ”No, it’s not a localhost.”
```

Те, що ліворуч від знака рівності, рівне тому, що праворуч. А якщо так, то ці два коди еквівалентні:

```
main :: IO ()
main = putStrLn (checkLocalhost ”173.194.22.100”)
```

```
main :: IO ()
main =
  putStrLn (if ”173.194.22.100” == ”127.0.0.1” ||
                ”173.194.22.100” == ”0.0.0.0”
              then ”it’s a localhost!”
              else ”No, it’s not a localhost.”)
```

Ми просто замінили застосування функції `checkLocalhost` її внутрішнім виразом, підставивши замість аргументу `ip` конкретний ря-

док `173.194.22.100`. В результаті, у залежності від істинності або хибності перевірок на рівність, ця умовна конструкція буде замінена одним з двох виразів. В цьому і полягає ідея: значення, що повертається функцією — це її останній, підсумковий вираз. Тобто якщо вираз:

```
"173.194.22.100" == "127.0.0.1" ||
"173.194.22.100" == "0.0.0.0"
```

дасть нам результат `True`, то ми переходимо до виразу з логічної гілки `then`. Якщо ж воно дасть нам `False` — ми переходимо до виразу з логічної гілки `else`. Це дає нам право стверджувати, що умовна конструкція виду:

```
if True
  then "it's a localhost!"
  else "No, it's not a localhost."
```

може бути замінена на перший нередукований вираз, рядок `it's a localhost!`, а умовну конструкцію виду:

```
if False
  then "it's a localhost!"
  else "No, it's not a localhost."
```

можна спокійно замінити другим нередукованим виразом, рядком `No, it's not a localhost.`. Тому код:

```
main :: IO ()
main = putStrLn (checkLocalhost "0.0.0.0")
```

еквівалентний кодові:

```
main :: IO ()
main = putStrLn ”it’s a localhost!”
```

Аналогічно, код:

```
main :: IO ()
main = putStrLn (checkLocalhost ”173.194.22.100”)
```

є ніщо інше, як:

```
main :: IO ()
main = putStrLn ”No, it’s not a localhost.”
```

Яким би складним не було логічне розгалуження всередині функції `checkLocalhost`, в кінцевому підсумку воно поверне/обчислить якийсь один підсумковий вираз. Саме тому з функції в Haskell не можна вийти в довільному місці, як це прийнято в імперативних мовах, адже вона не є набором інструкцій, вона є виразом, що складається з інших виразів. Ось чому функції в Haskell так просто компонувати одну з одною, і пізніше ми зустрінемо безліч таких прикладів.

Для допитливих

Уважний читач, безсумнівно помітив незвичайне оголошення головної функції нашого проекту, функції `main`:

```
main :: IO () -- Оголошення?
main = putStrLn ...
```

Якщо `IO` — це тип, що таке `()`? І чому вказаний лише один тип? Що таке `IO ()`: аргумент функції `main`, або ж те, що вона обчислює? Шкодую, але поки я змушений зберегти це в таємниці. Коли ми ближче познайомимося з Другим Китом Haskell, я неодмінно розповім про цей дивний `IO ()`.

Розділ 8

Вибір і зразки

Цей розділ відкриє нам інші способи вибору, а також познайомить нас зі зразками. Запевняю, ви закохаєтесь в них!

Не тільки з двох

Часто ми хочемо вибирати не тільки з двох можливих варіантів. Ось як це можна зробити:

```
analyzeGold :: Int -> String
analyzeGold standard =
  if standard == 999
    then ”Wow! 999 standard!”
    else if standard == 750
      then ”Great! 750 standard.”
      else if standard == 585
        then ”Not bad! 585 standard.”
        else ”I don’t know such a standard...”

main :: IO ()main = putStrLn (analyzeGold 999)
```

Впевнений, що ви вже стираєте плювок з екрану. Вкладена `if-then-else` конструкція не може нікому сподобатися, адже вона вкрай незручна в зверненні. А якщо аналізованих проб золота було б штук п’ять або сім, ця драбина стала б справді жахливою. На щастя, в Haskell можна написати по-іншому:

```
analyzeGold :: Int -> String
analyzeGold standard =
  if | standard == 999 -> ”Wow! 999 standard!”
     | standard == 750 -> ”Great! 750 standard.”
     | standard == 585 -> ”Not bad! 585 standard.”
     | otherwise -> ”I don’t know such a standard...”
```

Так красивіше, чи не так? Це — множинний `if`. Працює він за схемою:

```
if | CONDITION1 -> EXPR1
   | CONDITION2 -> EXPR2
   | ...
   | CONDITIONn -> EXPRn
   | otherwise -> COMMON_EXPRESSION
```

де `CONDITION1..n` — вирази, що дають хибу або істину, а `EXPR1..n` — відповідні їм результуючі вирази. Особлива функція `otherwise` відповідає загальному випадку, коли жоден з логічних виразів не дав `True`, і в цій ситуації результатом умовної конструкції стане вираз `COMMON_EXPRESSION`.

Не нехтуйте `otherwise`! Якщо ви його не вкажете і при цьому застосуйте функцію `analyzeGold` до значення, відмінного від перевірених:

```
analyzeGold :: Int -> String
analyzeGold standard =
  if | standard == 999 -> "Wow! 999 standard!"
     | standard == 750 -> "Great! 750 standard."
     | standard == 585 -> "Not bad! 585 standard."

main :: IO ()
main = putStrLn (analyzeGold 583) -- Ой...
```

компіляція завершиться успішно, однак у момент запуску програми на вас чекає неприємний сюрприз у вигляді помилки:

```
Non-exhaustive guards in multi-way if
```

Перевірка вийшла неповною, от і отримаєте помилку. До речі, бачите слово `guards` в повідомленні про помилку? Вертикальні риски перед логічними виразами — це і є охоронці (англ. guard), які постійно охороняють наші умови. Дотепну назву вибрали. Щоб читати їх було легше, сприймайте їх як аналог слова «АБО».

А зараз стоп. Адже ви спробували зібрати цей код, чи не так? А чому ви не лаєтеся? Адже такий код не скомпілюється, так як не вистачає однієї маленької, але важливої деталі. Ось як має виглядати модуль `Main`:

```haskell
{-# LANGUAGE MultiWayIf #-} -- Що це??

module Main whereanalyzeGold :: Int -> String
analyzeGold standard =
  if | standard == 999 -> "Wow! 999 standard!"
     | standard == 750 -> "Great! 750 standard."
     | standard == 585 -> "Not bad! 585 standard."
     | otherwise -> "I don't know such a standard..."

main :: IO ()
main = putStrLn (analyzeGold 999)
```

Ось тепер все гаразд. Але що це за дивний коментар у першому рядку модуля? Начебто оформлений як багаторядковий коментар, але виглядає незвично. Перед нами — вказівка розширення мови Haskell. Стандарт Haskell 2010 — це офіційний стрижень мови. Однак компілятор GHC, давно вже став компілятором за замовчуванням при розробці на Haskell та володіє рядом особливих можливостей. За замовчуванням багато які з цих можливостей вимкнені, а прагма `LANGUAGE` якраз для того і призначена, щоб їх вмикати/активізувати. В даному випадку ми увімкнули розширення `MultiWayIf`. Саме це розширення дозволяє нам використовувати множинний `if`. Такого роду розширень існує дуже багато, і ми будемо часто їх використовувати. Пам'ятайте: розширення, включене з допомогою прагми `LANGUAGE`, діє лише в рамках поточного модуля. І якщо я прописав його тільки в модулі `app/Main.hs`, то на модуль `src/Lib.hs` механізм `MultiWayIf` не поширюється.

Без Якщо

Множинний `if` досить зручний, але є спосіб більш красивий. Погляньте:

```
analyzeGold :: Int -> String
analyzeGold standard
  | standard == 999 = "Wow! 999 standard!"
  | standard == 750 = "Great! 750 standard."
  | standard == 585 = "Not bad! 585 standard."
  | otherwise = "I don't know such a standard..."
```

Ключове слово `if` зникло. Схема тут така:

```
function arg -- Немає знаку рівності?
  | CONDITION1 = EXPR1
  | CONDITION2 = EXPR2
  | ...
  | CONDITIONn = EXPRn
  | otherwise  = COMMON_EXPRESSION
```

Структура майже така ж, але, крім зникнення ключового слова `if`, ми тепер використовуємо знаки рівності замість стрілок. Саме тому зник знайомий нам знак рівності після імені аргументу `arg`. Насправді він, звісно, нікуди не зник, він лише перейшов у вирази. А щоб це легше прочитати, запишемо вирази в рядок:

```
function arg |   CONDITION1 =         EXPR1   | ...
```

```
ця                    або            дорівнює
функція                                        цьому
                                               виразу
                          у разі
                          істинності
                          цього
                          виразу
                                                      або і т. д.
```

Тобто перед нами вже не одне визначення функції, а ланцюжок визначень, тому нам і не потрібно ключове слово `if`. Але і цей ланцюжок визначень можна спростити.

Порівняння зі зразком

Прибравши слово `if`, ми і з нашими віртуальними «АБО» можемо розлучитися. В цьому випадку залишиться лише це:

```
analyzeGold :: Int -> String -- Одне оголошення.
-- Та безліч визначень...
analyzeGold 999 = "Wow! 999 standard!"
analyzeGold 750 = "Great! 750 standard."
analyzeGold 585 = "Not bad! 585 standard."
analyzeGold _ = "I don't know such a standard..."
```

Ми просто перерахували визначення функції `analyzeGold` одне за іншим. На перший погляд, можливість безлічі визначень однієї і тієї ж функції дивує, але якщо згадати, що застосування функції це вираз, тоді нічого дивного. Ось як це читається:

```
      analyzeGold 999             =        "Wow! 999 standard!"

якщо ця функція  застосовується  тоді     цьому виразу
                 до цього        вона
                 аргументу       дорівнює

      analyzeGold 750             =        "Wow! 999 standard!"

якщо ця функція  застосовується  тоді     іншому виразу
                 до іншого       вона
                 аргументу       дорівнює
...
           analyzeGold _ =        "I don't know such a standard..."
в
іншому    ця функція   просто   деякому загальному виразу
ж випадку              дорівнює
```

Коли функція `analyzeGold` застосовується до конкретного аргументу, цей аргумент послідовно порівнюється зі зразком (англ. pattern matching). Зразків тут три: `999`, `750` і `585`. І якщо раніше ми порівнювали аргумент з цими числовими значеннями явно, за допомогою функції `==`, тепер це відбувається приховано. Ідея порівняння зі зразком дуже проста: щось (в даному випадку реальний аргумент) зіставляється зі зразком (або зразками) на предмет «підходить/не підходить». Якщо підходить — тобто порівняння зі зразком дає результат `True` —готово, використовуємо відповідний вираз. Якщо ж не підходить — переходимо до наступного зразка.

Порівняння зі зразком використовується в Haskell надзвичайно широко. В україномовній літературі переклад словосполучення «pattern matching» не особливо закріпився, замість цього так і кажуть «патерн матчінг». Я зроблю так само.

Але що це за символ підкреслення такий, в останньому варіанті визначення? Ось цей:

```
analyzeGold _ = ”I don’t know such a standard...”
```

З формальної точки зору, це — універсальний зразок, порівняння з яким завжди істинно (ще говорять, що з ним матчиться (англ. match) все що завгодно). А з неформальної — це символ, який можна прочитати, як «мені все одно». Ми неначе кажемо: «У даному разі нас не цікавить конкретний вміст аргументу, нам все одно, ми просто повертаємо рядок `I don’t know such a standard...`».

Важливо відзначити, що порівняння аргументу зі зразками відбувається послідовно, згори вниз. Тому якщо ми напишемо так:

```
analyzeGold :: Int -> String
analyzeGold _ = ”I don’t know such a standard...”
analyzeGold 999 = ”Wow! 999 standard!”
analyzeGold 750 = ”Great! 750 standard.”
analyzeGold 585 = ”Not bad! 585 standard.”
```

наша функція буде завжди повертати перший вираз, рядок `I don’t know such a standard...`, і це цілком очікувано: перша ж перевірка гарантовано дасть нам `True`, адже з зразком _ збігається все що завгодно. Таким чином, загальний зразок слід розташовувати в самому кінці, щоб ми потрапили на нього лише після того, як не спрацювали всі інші зразки.

case

Існує ще один вид патерн матчінгу, за допомогою конструкції `case-of`:

```
analyzeGold standard =
  case standard of
    999 -> "Wow! 999 standard!"
    750 -> "Great! 750 standard."
    585 -> "Not bad! 585 standard."
    _ -> "I don't know such a standard..."
```

Запам'ятайте конструкцію `case-of`, ми зустрінемося з нею не раз. Працює вона згідно моделі:

```
case ВИРАЗ of
  PATTERN1 -> EXPR1
  PATTERN2 -> EXPR2
  ...
  PATTERNn -> EXPRn
  _ -> COMMON_EXPRESSION
```

де `EXPRESSION` — аналізований вираз, послідовно порівнюється із зразками `PATTERN1..n`. Якщо жоден не спрацював — як звичайно, впираємося в універсальний зразок `_` і видаємо `COMMON_EXPRESSION`.

У наступних розділах ми зустрінемося й з іншими видами патерн матчінгу, адже він використовується не лише для вибору.

Розділ 9

Нехай буде там, Де…

У цьому розділі ми дізнаємося, як зробити наші функції більш зручними й читабельними.

Нехай

Розглянемо наступну функцію:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  if | timeInS < 40 -> timeInS + 120
     | timeInS >= 40 -> timeInS + 8 + 120
```

Ми рахуємо час деякої події, і якщо початкове значення менше `40` секунд — результуючий час збільшується на `120` секунд, в іншому випадку — ще на `8` секунд більше. Перед нами класичний приклад «магічних чисел» (англ. magic numbers), коли зміст конкретних значень приховано за сімома печатями. Що за `40` та `8`? Щоб уникнути

цієї проблеми можна ввести тимчасові вирази, і тоді код стане зовсім іншим:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40
      correction = 120
      delta = 8
  in if | timeInS < threshold -> timeInS + correction
        | timeInS >= threshold -> timeInS + delta + correction
```

Ось, зовсім інша справа! Ми позбулися від «магічних чисел», ввівши пояснючі вирази `threshold`, `correction` і `delta`. Конструкція `let-in` вводить пояснювальні вирази за схемою:

```
let DECLARATIONS in EXPRESSION
```

де `DECLARATIONS` — вирази, декларовані нами, а `EXPRESSION` — вираз, в якому використовується вирази з `DECLARATION`. Коли ми написали:

```
let threshold = 40
```

ми оголосили: «Відтепер вираз `threshold` дорівнює виразу `40`». Виглядає як присвоювання, але ми вже знаємо, що в Haskell його немає. Тепер вираз `threshold` може замінити собою число `40` всередині виразу, що йде за словом `in`:

```
let threshold = 40
    ...
in if | timeInS < threshold -> ...
      | timeInS >= threshold -> ...
```

Ця конструкція легко читається:

```
let   threshold =           40 ... in ...

нехай цей       буде        цьому  в тому
      вираз     дорівнювати виразу виразі
```

З допомогою ключового слова `let` можна ввести скільки завгодно пояснювальних/проміжних виразів, що робить наш код, по-перше, зрозумілішим, а по-друге, коротшим. Так, у цьому конкретному випадку код став трохи довшим, але в наступних розділах ми побачимо ситуацію, коли проміжні значення скорочують код в рази.

Важливо пам'ятати, що введені конструкцією `let-in` вирази існують лише в рамках виразу, що слідує за словом `in`. Змінимо функцію:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40
    correction = 120
  in if | timeInS < threshold -> timeInS + correction
        | timeInS >= threshold ->
          let delta = 8 in timeInS
                           + delta
                           + correction

              це           існує лише в
              вираз        рамках цього виразу
```

Ми скоротили область видимості проміжного виразу `delta`, зробивши його видимим лише у виразі `timeInS + delta + correction`.

При бажанні `let`-вирази можна записувати і в рядок:

```
...
let threshold = 40; correction = 120
in if | timeInS < threshold -> timeInS + correction
      | timeInS >= threshold ->
        let delta = 8 in timeInS + delta + correction
```

У цьому випадку ми перераховуємо їх через крапку з комою. Особисто мені такий стиль не подобається, але вибирати вам.

Де

Існує інший спосіб введення проміжних виразів:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  if | timeInS < threshold -> timeInS + correction
     | timeInS >= threshold -> timeInS + delta + correction
  where
    threshold = 40
    correction = 120
    delta = 8
```

Ключове слово `where` робить приблизно те ж, що й `let`, але проміжні вирази задаються в кінці функції. Така конструкція читається подібно до наукової формули:

```
  S = V * t, -- Вираз
де
  -- Все те, що
  -- використовується
  -- у виразі.
  S = відстань,
  V = швидкість,
  t = час.
```

На відміну від `let`, яке може бути використане для введення суперлокального виразу (як в останньому прикладі з `delta`), `where`-вирази доступні в будь-якій частині виразу, що передує ключовому слову `where`.

Разом

Ми можемо використовувати `let-in` і `where` спільно, в рамках однієї функції:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40 in
  if | timeInS < threshold -> timeInS + correction
     | timeInS >= threshold -> timeInS + delta + correction
  where
    correction = 120
    delta = 8
```

Частина проміжних значень задана вгорі, а частина — внизу. Загальна рекомендація: не змішуйте `let-in` і `where` без особливої потреби, такий код читається важко, виглядає надлишковим.

Зазначу, що в якості проміжних можуть виступати і більш складні вирази. Наприклад:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let threshold = 40 in
  if | timeInS < threshold -> timeInS + correction
     | timeInS >= threshold -> timeInS + delta + correction
  where
    -- Цей проміжний вираз залежить від аргументу...
    correction = timeInS * 2
    -- А цей - від іншого виразу...
    delta = correction - 4
```

Вираз `correction` дорівнює `timeInS * 2`, тобто тепер він залежить від значення аргументу функції. А вираз `delta` залежить у свою чергу від `correction`. Причому ми можемо змінювати порядок задання виразів:

```
...
let threshold = 40
in if | timeInS < threshold -> timeInS + correction
      | timeInS >= threshold -> timeInS + delta + correction
where
  delta = correction - 4
  correction = timeInS * 2
```

Вираз `delta` тепер задано першим, але це не має ніякого значення. Адже ми лише оголошуємо рівності, і результат цих оголошень не впливає на кінцевий результат обчислень. Звісно, порядок оголошення рівностей не важливий і для `let`-виразів:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let delta = correction - 4
      threshold = 40
  in if | timeInS < threshold -> timeInS + correction
        | timeInS >= threshold -> timeInS + delta + correction
  where
    correction = timeInS * 2
```

Мало того, що ми поставили `let`-вирази в іншому порядку, так ми ще й використовували в одному з них вираз `correction`! Тобто в `let`-виразі використовувався `where`-вираз. А ось виконати зворотне, на жаль, не вийде:

```
calculateTime :: Int -> Int
calculateTime timeInS =
  let delta = correction - 4
     threshold = 40
  in
  if | timeInS < threshold -> timeInS + correction
     | timeInS >= threshold -> timeInS + delta + correction
  where
    correction = timeInS * 2 * threshold -- 3 let??
```

При спробі скомпілювати такий код ми отримаємо помилку:

```
Not in scope: 'threshold'
```

Обмеження таке: використовувати `let`-вирази всередині `where`-виразів неможливо, адже останні вже не входять у вираз, що йде за словом `in`.

Ну що ж, пора рухатися далі, адже вміст наших функцій не обмежений умовними конструкціями.

Розділ 10

Світ Операторів

Оператор (англ. operator) — особливий випадок функції. У попередніх розділах ми вже познайомилися з ними, залишилося пояснити докладніше.

Згадаймо наш найперший вираз:

```
1 + 2
```

Функція `+` записана в інфіксній (англ. infix) формі, тобто між своїми аргументами. Такий запис виглядає природніше, ніж звичайний:

```
(+) 1 2
```

Бачите круглі дужки? Вони говорять про те, що дана функція призначена для інфіксного запису. Автор цієї функції спочатку розраховував на інфіксну форму використання `1 + 2`, а не на звичайну `(+) 1 2`, саме тому ім'я функції у визначенні обрамлене в круглі дужки:

```
(+) :: ...
```

Функції, призначені для інфіксної форми застосування, називають операторами.

Якщо ж ім'я функції не обрамлене в круглі дужки, мається на увазі, що ми розраховуємо на звичайну форму її застосування. Однак і в цьому випадку можна застосовувати її інфіксно, але ім'я повинно лежати в зворотніх одинарних лапках (англ. backtick).

Визначимо функцію `isEqualTo`, що є аналогом оператора перевірки на рівність для двох цілочисельних значень:

```
isEqualTo :: Int -> Int -> Bool
isEqualTo x y = x == y
```

При звичайній формі її застосування виглядало би так:

```
  ...
  if isEqualTo code1 code2 then ... else ...
  where code1 = 123
        code2 = 124
...
```

Але давайте перепишемо у інфіксній формі:

```
  ...
  if code1 `isEqualTo` code2 then ... else ...
  where code1 = 123
        code2 = 124
...
```

Набагато краще, адже тепер код читається як звичайний англійський текст:

```
...
if code1 `isEqualTo` code2 ...
if code1 is equal to code2 ...
...
```

Строго кажучи, назва «оператор» вельми умовна, ми можемо її й не використовувати. Говорити про функції додавання настільки ж коректно, як і про оператор додавання.

Навіщо це потрібно?

Майже всі ASCII-символи (а також їх різноманітні комбінації) можна використовувати в якості операторів в Haskell. Це дає нам широкі можливості для реалізації різних EDSL (англ. Embedded Domain Specific Language), свого роду «мов у мові». Ось приклад:

```
div ! class_ "nav-wrapper" $
  a ! class_ "brand-logo sans" ! href/" $
    "#ohaskell"
```

Кожен, хто знайомий з веб-розробкою, миттєво впізнає в цьому коді HTML. Цей шматочок коду будує HTML-шаблон для веб-варіанту даної книги. Те, що ви бачите — це абсолютно легальний Haskell-код, в процесі роботи якого генерується реальний HTML: тег `<div>` з класом `nav-wrapper`, всередині якого лежить `<a>`-посилання з двома класами, кореневою адресою та внутрішнім текстом `#ohaskell`.

Ідентифікатори `div`, `class_` і `href` — це імена функцій, а символи `!` і `$` — це оператори, записані в інфіксній формі. Найголовніше, що для розуміння цього коду нам абсолютно необов'язково знати, де визначені всі ці функції/оператори і як вони працюють. Це важли-

ва думка, яку я неодноразово буду повторювати у наступних розділах:

> Щоб використовувати функції, нам зовсім необов'язково знати їх внутрішню будову.

А про EDSL пам'ятайте, ми з ними ще зустрінемося.

Розділ 11

Список

Пам'ятаєте, в одному з попередніх розділів я говорив, що познайомлю вас ще з кількома стандартними типами даних в Haskell? Настав час дізнатися про списки.

Список (англ. list) — це стандартний тип, що характеризує вже не просто дані, а структуру даних (англ. data structure). Ця структура є набором даних одного типу, і навряд чи хоч одна реальна Haskell-програма може обійтися без списків.

Структури, що містять дані одного типу, називають гомогенними (в перекладі з грецької: «одного роду»).

Ось список з трьох цілих чисел:

```
[1, 2, 3]
```

Квадратні дужки і значення, розділені комами. Ось так виглядає список з двох значень типу `Double`:

```
[1.3, 45.7899]
```

а ось і список з одного-єдиного символу:

```
['H']
```

або ось з чотирьох рядків, що відображають імена протоколів транспортного рівня OSI-моделі:

```
["TCP", "UDP", "DCCP", "SCTP"]
```

Якщо у вас є досвід розробки на мові C, ви можете подумати, що список схожий на масив. Однак, хоча подібності є, я навмисно уникаю слова «масив», тому що в Haskell існують масиви (англ. array), а це дещо інша структура даних.

Список — це теж вираз, тому можна легко створити список списків довільної вкладеності. Ось так буде виглядати список з ряду протоколів трьох рівнів OSI-моделі:

```
[ ["DHCP", "FTP", "HTTP"]
, ["TCP", "UDP", "DCCP", "SCTP"]
, ["ARP", "NDP", "OSPF"]
]
```

Це список списків рядків. Форматування щодо квадратних дужок досить вільне, при бажанні можна написати і так:

```
[["DHCP", "FTP", "HTTP" ],
["TCP", "UDP", "DCCP", "SCTP"],
["ARP", "NDP", "OSPF" ]]
```

Список може бути порожнім, тобто не містити в собі ніяких даних:

```
[]
```

Тип списку

Оскільки список є структурою, що містить дані деякого типу, який же тип самого списку? Ось:

```
[Int] -- Список цілих чисел
[Char] -- Список символів
[String] -- Список рядків
```

Тобто тип списку так і вказується у квадратних дужках. Згаданий раніше список списків рядків має такий вигляд:

```
[[String]] -- Список списків рядків
```

Модель дуже проста:

```
[   [String]  ]

    | Тип  |
    |даних |

|  Тип        |
|  списку     |
|_ цих даних _|
```

Зберігати дані різних типів в стандартному списку неможливо. Однак незабаром ми познайомимося з іншою стандартною структурою даних, яка це дозволяє.

Дії над списками

Якщо списки створюються — значить це комусь потрібно. Зі списком можна робити дуже багато всього. У стандартній Haskell-бібліотеці існує окремий модуль `Data.List`, що включає широкий набір функцій, що працюють зі списком. Відкриємо модуль `Main` та імпортуємо в нього модуль `Data.List`:

```
module Main where

-- Стандартний модуль для роботи зі списками.
import Data.List

main :: IO ()
main = putStrLn (head [”Vim”, ”Emacs”, ”Atom”])
```

Функція `head` повертає голову списку, тобто його перший елемент. При запуску цієї програми на виході отримаємо:

```
Vim
```

Модель така:

```
[”Vim”  , ”Emacs”, ”Atom”]

 голова   |___ хвіст ___|
```

Така собі гусениця виходить: перший елемент — голова, а все інше — хвіст. Функція `tail` повертає хвіст:

```
main :: IO ()
main = print (tail [”Vim”, ”Emacs”, ”Atom”])
```

Ось результат:

```
["Emacs", "Atom"]
```

Функція `tail` формує інший список, який представляє собою все від початкового списку, крім голови. Зверніть увагу на нову функцію `print`. В даному випадку ми не могли б використовувати нашу знайому `putStrLn`, адже вона застосовується до значень типу `String`, в той час як функція `tail` поверне нам значення типу `[String]`. Адже ми пам'ятаємо про строгість компілятора: що очікуємо, то й повинні отримати. Функція `print` призначена для «стрінгіфікації» значень: вона бере значення деякого типу і виводить це значення на консоль у вигляді рядка.

Уважний читач запитає, яким же чином функція `print` дізнається, як саме відобразити конкретне значення у вигляді рядка? О, це цікава тема, але вона відноситься до Третього Кита Haskell, до знайомства з яким нам ще далеко.

Можна отримати довжину списку:

```
handleTableRow :: String -> String
handleTableRow row
  | length row == 2 = composeTwoOptionsFrom row
  | length row == 3 = composeThreeOptionsFrom row
  | otherwise = invalidRow row
```

Це трішки видозмінений шматочок однієї моєї програми, функція `handleTableRow` обробляє рядок таблиці. Стандартна функція `length` дає нам довжину списку (число елементів у ньому). В даному випадку ми дізнаємося кількість елементів у рядку таблиці `row`, і залежно від цієї довжини застосовуємо до цього рядка функцію `composeTwoOptionsFrom` або `composeThreeOptionsFrom`.

Але стривайте, а де ж тут список? Функція `handleTableRow` застосовується до рядка і обчислює рядок. А вся справа в тому, що рядок є

ніщо інше, як список символів. Тобто тип `String` еквівалентний типу `[Char]`. Скажу більше: `String` — це навіть не самостійний тип, це всього лише псевдонім для типу `[Char]`, і ось як він заданий:

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

Ключове слово `type` вводить псевдонім для вже існуючого типу. Читається так:

```
type String =        [Char]

тип  цей    дорівнює тому
```

Таким чином, оголошення функції `handleTableRow` можна було б переписати так:

```
handleTableRow :: [Char] -> [Char]
```

При роботі зі списками ми можемо використовувати вже знайомі проміжні вирази, наприклад:

```
handleTableRow :: String -> String
handleTableRow row
  | size == 2 = composeTwoOptionsFrom row
  | size == 3 = composeThreeOptionsFrom row
  | otherwise = invalidRow row
  where size = length row
```

А можна й так:

```
handleTableRow :: String -> String
handleTableRow row
  | twoOptions = composeTwoOptionsFrom row
  | threeOptions = composeThreeOptionsFrom row
  | otherwise = invalidRow row
  where
    size = length row -- Дізнаємося довжину
    twoOptions = size == 2 -- порівнюємо ...
    threeOptions = size == 3 -- ... і ще раз
```

Тут вирази `twoOptions` і `threeOptions` мають вже знайомий нам стандартний тип `Bool`, адже вони дорівнюють результату порівняння значення `size` з числом.

Незмінність списку

Як ви вже знаєте, всі дані в Haskell незмінні, як Єгипетські піраміди. Списки — не виняток: ми не можемо змінити існуючий список, ми можемо лише створити на його основі новий список. Наприклад:

```
addTo :: String -> [String] -> [String]
addTo newHost hosts = newHost : hosts

main :: IO ()
main = print (”124.67.54.90” `addTo` hosts)
where hosts = [”45.67.78.89”, ”123.45.65.54”]
```

Результат цієї програми такий:

```
[”124.67.54.90”,”45.67.78.89”,”123.45.65.54”]
```

Розглянемо визначення функції `addTo`:

```
addTo newHost hosts = newHost : hosts
```

Стандартний оператор `:` додає значення, яке є лівим операндом, на початок списку, який є правим операндом. Читається так:

```
newHost :        hosts

        цей
        оператор

бере
це
значення
                 і додає
                 його на початок
                 цього списку
```

Природно, тип значення зліва зобов'язаний співпадати з типом значень, що містяться в списку праворуч.

З концептуальної точки зору функція `addTo` додала нову IP-адресу в початок списку `host`. У дійсності ж ніякого додавання не сталося, бо списки незмінні. Оператор `:` взяв значення `newHost` та список `host` і створив на їхній основі новий список, який містить в собі вже три IP-адреси замість двох.

Переліки

Припустимо, що нам знадобився список цілих чисел від одного до десяти. Пишемо:

```
main :: IO ()
main = print tenNumbers
  where tenNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Непогано, але надмірно, адже чисел могло бути сто, тисяча. Є кращий шлях:

```
main :: IO ()
main = print tenNumbers
  where tenNumbers = [1..10]
```

Красиво, чи не так? Вираз у квадратних дужках називається переліком (англ. enumeration або скорочено enum). Іноді його називають також арифметичною послідовністю. Ідея гранично проста: навіщо вказувати вміст списку цілком у тій ситуації, коли можна вказати лише діапазон значень? Це ми і зробили:

```
[1..10] = [1,2,3,4,5,6,7,8,9,10]
```

Значення зліва від `..` — це початок діапазону, а значення праворуч — його кінець. Компілятор сам здогадається, що крок між числами в даній послідовності дорівнює 1. Ось ще приклад:

```
[3..17] = [3,4,5,6,7,8,9,10,11,12,13,14,15,16,17]
```

```
–          –

    ==                                    ==
```

Ми можемо задати крок і явно:

```
[2,4..10] = [2,4,6,8,10]
```

Отримали тільки парні значення. Схема проста:

```
[2,       4 ..   10    ]

перший           кінець
         другий

|  різниця   |
|_ дає крок _|
```

Ось ще приклад:

```
[3,9..28] = [3,9,15,21,27]
```

Можна задати і спадний діапазон:

```
[9,8..1] = [9,8,7,6,5,4,3,2,1]
```

Або так:

```
[-9,-8..-1] = [-9,-8,-7,-6,-5,-4,-3,-2,-1]
```

Так, негативні числа теж працюють. Можна взяти також і числа з плаваючою крапкою:

```
[1.02,1.04..1.16] = [1.02,1.04,1.06,1.08,1.1,1.12,1.14,1.16]
```

Загалом, ідея зрозуміла. Але що це ми весь час з числами та з числами! Візьмемо символи:

```
['a'..'z'] = "abcdefghijklmnopqrstuvwxyz"
```

Діапазон від `a` до `z` — отримали англійський алфавіт у вигляді `[Char]` або, як ми вже знаємо, просто `String`. При великому бажанні явно

задати крок можна і тут:

```
['a','c'..'z'] = "acegikmoqsuwy"
```

Ось така краса.

Тепер, після знайомства зі списками, ми будемо використовувати їх постійно.

Для допитливих

У розділі про діапазони для списку ми оперували значеннями типу `Int`, `Double` і `Char`. Виникає питання: а чи можна використовувати значення якихось інших типів? Відповідаю: можна, але зі застереженням. Спробуємо зробити це з рядком:

```
haskell main :: IO () main = print ["a","aa"..."aaaaaa"] -- Ну-ну...
```

При спробі зібрати такий код побачимо помилку:

```
bash No for instance (Enum [Char]) arising from the arithmetic sequence
'"a", "aa" .. "aaaaaa"'
```

І дивуватися тут нічому: крок між рядками абсурдний, і компілятор розгублений. Не всі типи підходять для переліків через своєю природу, однак у майбутньому, коли ми навчимося створювати наші власні типи, ми дізнаємося, що їх цілком можна використовувати в діапазонах. Наберіться терпіння.

Розділ 12

Кортеж

У цьому розділі ми познайомимося з кортежем і ще ближче подружимося з патерн матчінгом.

Кортеж (англ. tuple) — ще одна стандартна структура даних, але, на відміну від списку, вона може містити як дані одного типу, так і різних.

Структури, здатні містити дані різних типів, називають гетерогенними (в перекладі з грецької: «різного роду»).

Ось як виглядає кортеж:

```
("Haskell", 2010)
```

Круглі дужки і значення розділені комами. Цей кортеж містить значення типу `String` і ще одне, типу `Int`. Ось ще приклад:

```
("Haskell", "2010", "Standard")
```

Тобто ніщо не заважає нам зберігати в кортежі дані одного типу.

Тип кортежу

Тип списку рядків, як ви пам'ятаєте, `[String]`. І не важливо, скільки рядків ми запхали в список, один або мільйон — його тип залишиться незмінним. З кортежем же абсолютно-по іншому.

Тип кортежу залежить від кількості його елементів. Ось тип кортежу, що містить два рядки:

```
(String, String)
```

Ось ще приклад:

```
(Double, Double, Int)
```

І ще:

```
(Bool, Double, Int, String)
```

Тип кортежу явно відображає його вміст. Тому якщо функція застосовується до кортежу з двох рядків, застосувати її до кортежу з трьох ніяк не вийде, адже типи цих кортежів відрізняються:

```
-- Різні типи
(String, String)
(String, String, String)
```

Дії над кортежами

Зі списками можна робити багато всього, а от з кортежами — не дуже. Найчастіші дії — власне формування кортежу і добування даних. Наприклад:

```
makeAlias :: String -> String -> (String, String)
makeAlias host alias = (host, alias)
```

Мабуть, нічого простішого придумати не можна: на вході два аргументи, на виході — двоелементний кортеж з цими аргументами. Двоелементний кортеж називають ще парою (англ. pair). І хоча кортеж може містити скільки завгодно елементів, на практиці саме пари зустрічаються найчастіше.

Зверніть увагу, наскільки легко створюється кортеж. Причина цього — вже знайомий нам патерн матчінг:

```
makeAlias host alias = (host, alias)

           ____          ____

                =====          =====
```

Ми просто вказуємо відповідність між лівою і правою сторонами визначення: «Нехай перший елемент пари дорівнює аргументу `host`, а другий — аргументу `alias`». Нічого зручнішого і простішого й придумати не можна. І якщо б ми хотіли отримати кортеж з трьох елементів, це виглядало б так:

```
makeAlias :: String -> String -> (String, String, String)
makeAlias host alias = (host, "https://" ++ host, alias)

           ____          ____                ____

                =====                             =====
```

Оператор `++` — це оператор конкатенації, склеює два рядки в один. Строго кажучи, він склеює два списки, але ми-то з вами вже зна-

ємо, що `String` є ніщо інше, як `[Char]`. Таким чином, `"https://" ++ "www.google.com"` дає нам `"https://www.google.com"`.

Витяг елементів з кортежу також проводиться через патерн матчінг:

```
main :: IO ()
main =
  let (host, alias) = makeAlias "173.194.71.106"
                                "www.google.com"
  in print (host ++ ", " ++ alias)
```

Функція `makeAlias` дає нам пару з хоста й імені. Але що це за дивний запис біля вже знайомого нам слова `let`? Це проміжний вираз, але вираз хитрий, утворений через патерн матчінг. Щоб було зрозуміліше, спочатку перепишемо функцію без нього:

```
main :: IO ()
main =
  let pair = makeAlias "173.194.71.106"
                       "www.google.com"
      host = fst pair -- Беремо перше...
      alias = snd pair -- Беремо друге...
  in print (host ++ ", " ++ alias)
```

При запуску цієї програми отримаємо:

```
"173.194.71.106, www.google.com"
```

Стандартні функції `fst` і `snd` повертають перший та другий елемент кортежу відповідно. Вираз `pair` відповідає парі, вираз `host` — значенню хоста, а `alias` — значенню імені. Але чи не здається вам такий спосіб надмірним? Ми в Haskell любимо витончені рішення, тому віддаємо перевагу патерн матчінгу. Ось як виглядає вищенаведений спосіб:

```
let (host, alias) = makeAlias ”173.194.71.106” ”www.google.com”
let (host, alias) = (”173.194.71.106”, ”www.google.com”)

                     дане значення
     це
     хост
                                        а ось це значення
          це
          ім’я
```

Ось така проста магія. Функція `makeAlias` дає нам пару, і ми достовірно знаємо це! А якщо знаємо, нам не потрібно вводити проміжні вирази на кшталт `pair`. Ми відразу кажемо:

```
let (host, alias) = makeAlias ”173.194.71.106” ”www.google.com”

                    ми точно знаємо, що вираз,
                    обчислений цією функцією
     це ось
     така пара
```

Це «дзеркальна» модель: через патерн матчінг формуємо:

```
-- Формуємо праву сторону
-- на основі лівої...
host alias = (host, alias)

>>>>          >>>>

     >>>>>          >>>>>
```

і через нього ж отримуємо:

```
-- Формуємо ліву сторону
-- на основі правої...
(host, alias) = (”173.194.71.106”, ”www.google.com”)

 <<<<            <<<<<<<<<<<<<<<<

       <<<<<                        <<<<<<<<<<<<<<<<
```

Ось ще один приклад роботи з кортежем через патерн матчінг:

```
chessMove :: String
             -> (String, String)
             -> (String, (String, String))
chessMove color (from, to) = (color, (from, to))

main :: IO ()
main = print (color ++ ”: ”++ from ++ ”-” ++ to)
  where
    (color, (from, to)) = chessMove ”white” (”e2”, ”e4”)
```

І на виході отримуємо:

```
”white: e2-e4”
```

Зверніть увагу, оголошення функції відформатоване трохи інакше: типи вишикувані один за одним через вирівнювання стрілок під двокрапкою. Ви часто зустрінете такий стиль в Haskell-проектах.

Функція `chessMove` дає нам кортеж з кортежем, а оскільки ми точно знаємо вигляд цього кортежу, відразу вказуємо `where`-вираз у вигляді зразка:

```
(color, (from, to)) = chessMove "white" ("e2", "e4")

 _____                          _______

        ====                           ====

              ..                             ....
```

До речі, я про це не згадав, але теоретично кортеж може складатися й з одного елемента:

```
useless :: String -> (String)
useless s = (s)

main :: IO ()
main = putStrLn s
  where (s) = useless "some"
```

Втім, враховуючи гетерогенність кортежу, мені важко уявити ситуацію, в якій одноелементний кортеж був би реально корисним.

Не всі

Ми можемо витягувати за зразком лише частину потрібної нам інформації. Пам'ятаєте універсальний зразок _? Погляньте:

```
-- Пояснюючі псевдоніми
type UUID = String
type FullName = String
type Email = String
type Age = Int
type Patient = (UUID, FullName, Email, Age)

patientEmail :: Patient -> Email
patientEmail (_, _, email, _) = email

main :: IO ()
main =
  putStrLn (patientEmail ( "63ab89d"
                         , "John Smith"
                         , "johnsm@gmail.com"
                         , 59
                         ))
```

Функція `patientEmail` дає нам пошту пацієнта. Тип `Patient` — це псевдонім для кортежу з чотирьох елементів: унікальний ідентифікатор, повне ім'я, адреса електронної пошти та вік. Додаткові псевдоніми роблять наш код зрозумілішим: одна справа бачити безликий `String` і зовсім інша — `Email`.

Розглянемо внутрішню будову функції `patientEmail`:

```
patientEmail (_, _, email, _) = email
```

Функція говорить нам: «Так, я знаю, що мій аргумент — це чотирьохелементний кортеж, але мене в ньому цікавить виключно третій елемент, відповідний поштовій адресі, його я і поверну». Універсальний зразок _ робить наш код лаконічнішим і зрозумілішим, адже він допомагає нам ігнорувати те, що нам нецікаво. Строго кажучи, ми не зобов'язані використовувати _, але з ним

буде краще.

А якщо помилилися?

При використанні патерн матчінгу щодо пари слід бути уважним. Уявімо собі, що вищезгаданий тип `Patient` був розширений:

```
type UUID = String
type FullName = String
type Email = String
type Age = Int
type DiseaseId = Int -- Новий елемент.
type Patient = ( UUID
               , FullName
               , Email
               , Age
               , DiseaseId
               )
```

Додано ідентифікатор захворювання. І все було б добре, але внести зміни у функцію `patientEmail` ми забули:

```
patientEmail :: Patient -> Email
patientEmail (_, _, email, _) = email

               ^  ^  ^      ^ -- А п'ятий де?
```

На щастя, в цьому випадку компілятор строго зверне нашу увагу на помилку:

```
Couldn't match type '(t0, t1, String, t2)'
with '(UUID, FullName, Email, Age, DiseaseId)'
Expected type: Patient
Actual type: (t0, t1, String, t2)
In the pattern: (_, _, email, _)
```

Воно й зрозуміло: функція `patientEmail` використовує зразок, який вже є некоректним. Ось чому при використанні патерн матчінгу слід бути уважним.

На цьому наше знайомство з кортежами вважаю завершеним, у наступних розділах ми будемо використовувати їх періодично.

Для допитливих

Для роботи з елементами багатоелементних кортежів можна використовувати готові бібліотеки, задля уникнення довгих патерн матчінгових ланцюжків. Наприклад, пакет tuple:

```
Data.Tuple.Select

main :: IO ()
main = print (sel4 (123, 7, "hydra", "DC:4", 44, "12.04"))
```

Функція `sel4` з модуля `Data.Tuple.Select` витягує четвертий елемент кортежу, в даному випадку рядок "`DC:4`". Там є функції аж до `sel32`, автори цілком розумно визнали, що ніхто, будучи в здоровому глузді, не стане оперувати кортежами, що складаються з більш ніж 32 елементів.

Крім того, ми й оновлювати елементи кортежу можемо:

```
import Data.Tuple.Update

main :: IO ()
main = print (upd2 2 (”si”, 45))
```

Природно, унаслідок незмінності кортежу, ніякого оновлення тут не відбувається, але виглядає симпатично. При запуску отримуємо результат:

```
(”si”,2)
```

Розділ 13

Лямбда-функція

Настав час познайомитися з важливою концепцією — лямбда-функцією. Саме з неї все і почалося. Приготуйтеся: в цьому розділі нас чекають нові відкриття.

Витоки

У далеких 1930-х молодий американський математик Алонзо Черч замислився над тим, що означає «обчислити». Плодом його роздумів стала система для формалізації поняття «обчислення», і назвав він її «лямбда-численням» (англ. lambda calculus, від імені грецької літери λ). В основі цієї системи лежить лямбда-функція, яку можна вважати «матір'ю функціонального програмування» в цілому і Haskell зокрема. Далі буду називати її ЛФ.

Щодо ЛФ можна сміливо сказати: «Все геніальне просто». Ідея ЛФ настільки корисна саме тому, що вона гранично проста. ЛФ — це анонімна функція. Ось як вона виглядає в Haskell:

```
\x -> x * x
```

Зворотний слеш на початку — ознака ЛФ. Порівняйте з математичною формою запису:

```
λx . x * x
```

Схоже, чи не правда? Сприймайте зворотний слеш у визначенні ЛФ як спинку літери λ.

ЛФ являє собою найпростіший вид функції, це така функція, роздягнена догола. У неї забрали не тільки оголошення, але й ім'я, залишивши лише необхідний мінімум у вигляді імен аргументів і внутрішнього виразу. Алонзо Черч зрозумів: щоб застосувати функцію, зовсім необов'язково її іменувати. І якщо у звичайній функції спочатку йде оголошення/визначення, а потім (десь) застосування з використанням імені, то у ЛФ все значно простіше: ми її визначаємо і тут же застосовуємо, на місці. Ось так:

```
(\x -> x * x) 5
```

Пам'ятаєте функцію `square`? Ось це її лямбда-аналог:

```
(\x -> x * x)      5

лямбда-абстракція  аргумент
```

Лямбда-абстракція (англ. lambda abstraction) — це особливий вираз, який породжує функцію, яку ми відразу ж застосовуємо до аргументу `5`. ЛФ з одним аргументом, як і просту функцію, ще називають «ЛФ від одного аргументу» чи «ЛФ одного аргументу». Також можна казати й «лямбда-абстракцію одного аргументу »

Побудова

Побудова лямбда-абстракції гранично проста:

```
\      x ->      x * x
ознака ім'я      вираз
ЛФ     аргументу
```

Відповідно, якщо ЛФ застосовується до двох аргументів — пишемо так:

```
\      x         y ->      x * y
ознака ім'я 1    ім'я 2    вираз
ЛФ     аргументу аргументу
```

І коли ми застосовуємо таку функцію:

```
(\x y -> x * y) 10 4
```

то просто підставляємо `10` на місце `x`, `4` — на місце `y`, і отримуємо вираз `10 * 4`:

```
(\x y -> x * y) 10 4
= 10 * 4
= 40
```

Загалом, все як зі звичайною функцією, навіть простіше.

Ми можемо ввести проміжне значення для лямбда-абстракції:

```
main :: IO ()
main = print (mul 10 4)
  where mul = \x y -> x * y
```

Тепер ми можемо застосовувати `mul` так само, ніби це сама лямбда-абстракція:

```
  mul 10 4
= (\x y -> x * y) 10 4
= 10 * 4
```

І тут ми наблизилися до одного важливого відкриття.

Тип функції

Ми знаємо, що у всіх даних в Haskell-програмі обов'язково є якийсь тип, який уважно перевіряється на етапі компіляції. Питання: який тип у виразу `mul` з попереднього прикладу?

```
where mul = \x y -> x * y -- Який тип?
```

Відповідь проста: тип `mul` такий же, як і у цієї лямбда-абстракції. З цього ми робимо важливий висновок: ЛФ має тип, як і звичайні дані. Але оскільки ЛФ є окремим випадком функції — значить і у звичайної функції теж є тип!

В нефункційних мовах між функціями і даними проведена чітка межа: ось це функції, а ось це — дані. Проте в Haskell між даними і функціями різниці немає, адже і те й інше спочиває на одній і тій же Черепасі. Ось тип функції `mul`:

```
mul :: a -> a -> a
```

Почекайте, скажете ви, але ж це оголошення функції! Абсолютно вірно: оголошення функції — це і є вказівка її типу. Пам'ятаєте, коли ми вперше познайомилися з функцією, я уточнив, що її оголо-

шення розділене подвійною двокрапкою? Так ось ця подвійна двокрапка і являє собою вказівку типу:

```
mul ::  a -> a -> a

ось має |   от    |
це  тип |_ такий _|
```

Точно так само ми можемо вказати тип будь-яких інших даних:

```
let coeff = 12 :: Double
```

Хоча ми знаємо, що в Haskell типи виводяться автоматично, іноді ми хочемо взяти цю турботу на себе. В даному випадку ми говоримо: «Нехай вираз `coeff` буде `12`, але тип його нехай буде `Double`, а не `Int`». Так само і з функцією: коли ми оголошуємо її — ми тим самим вказуємо її тип.

Але ви запитаєте, чи можемо ми не вказувати тип функції явно? Можемо:

```
square x = x * x
```

Це наша стара знайома, функція `square`. Коли вона буде застосована до значення типу `Int`, тип аргументу буде виведений автоматично як `Int`.

І оскільки функція характеризується типом так само, як й інші дані, ми робимо ще одне важливе відкриття: функціями можна оперувати як даними. Наприклад, можна створити список функцій:

```
main :: IO ()
main = putStrLn ((head functions) ”Hi”)
 where
   functions = [ \x -> x ++ ”val1”
               , \x -> x ++ ”val2”
               ]
```

Вираз `functions` — це список з двох функцій. Два лямбда-вирази породжують ці дві функції, але до моменту застосування вони нічого не роблять, вони неживі і марні. Але коли ми застосовуємо функцію `head` до цього списку, ми отримуємо перший елемент списку, тобто першу функцію. І отримавши, тут же застосовуємо цю функцію до рядка ”Hi”:

```
putStrLn ((head functions) ”Hi”)

           |   перша    |  її
           |   функція  |  аргумент
           |__ списку __|
```

Це рівносильно кодові:

```
putStrLn ((\x -> x ++ ”val1”) ”Hi”)
```

При запуску програми ми отримаємо:

```
Hi val1
```

До речі, а який тип списку `functions`? Його тип такий: `[String -> String]`. Тобто список функцій з одним аргументом типу `String`, які повертають значення типу `String`.

Локальні функції

Раз вже між ЛФ та простими функціями фактично немає розходжень, а функції є частковим випадком даних, ми можемо створювати функції локально для інших функцій:

```
-- Тут визначені функції
-- isInfixOf і isSuffixOf.
import Data.List

validComEmail :: String -> Bool
validComEmail email =
containsAtSign email && endsWithCom email
  where
    containsAtSign e = "@" `isInfixOf` e
    endsWithCom e = ".com" `isSuffixOf` e

main :: IO ()
main = putStrLn (if my validComEmail
  then "it's ok!"
  else "Non-com email!")
  where
    my = "haskeller@gmail.com"
```

Дещо наївна функція `validComEmail` перевіряє `.com`-адресу. Її вираз утворено оператором `&&` і двома виразами типу `Bool`. Ось як утворені ці вирази:

```
containsAtSign e = "@" `isInfixOf` e
endsWithCom e = ".com" `isSuffixOf` e
```

Це — дві функції, які ми визначили прямо в `where`-секції, тому вони існують тільки для основного виразу функції `validComEmail`. З простими функціями так роблять дуже часто: де вона потрібна, там її і визначають. Ми могли б написати і більш явно:

```
validComEmail :: String -> Bool
validComEmail email =
  containsAtSign email && endsWithCom email
  where
    -- Оголошуємо локальну функцію явно.
    containsAtSign :: String -> Bool
    containsAtSign e = "@" `isInfixOf` e
    -- І цю теж.
    endsWithCom :: String -> Bool
    endsWithCom e = ".com" `isSuffixOf` e
```

Втім, вказувати тип настільки простих функцій, як правило, необов'язково.

Ось як цей код виглядає з лямбда-абстракціями:

```
validComEmail :: String -> Bool
validComEmail email =
  containsAtSign email && endsWithCom email
  where
    containsAtSign = \e -> "@" `isInfixOf` e
    endsWithCom = \e -> ".com" `isSuffixOf` e
```

Тепер вирази `containsAtSign` і `endsWithCom` прирівняні до ЛФ від одного аргументу. У цьому випадку ми не вказуємо тип цих виразів. Втім, якщо дуже хочеться, можна вказати:

```
containsAtSign =
  (\e -> "@" `isInfixOf` e) :: String -> Bool


  лямбда-абстракція            тип цієї абстракції
```

Лямбда-абстракцію взято в дужки, щоб оголошення типу відноси-

лося до функції в цілому, а не тільки до аргументу `e`:

```
containsAtSign =
  \e -> "@" `isInfixOf` e :: String -> Bool

                              тип аргументу e,
                              а зовсім не всієї
                              функції!
```

Типу функції теж можна надати псевдонім:

```
-- Псевдонім типу функції.
type Func = String -> Bool

validComEmail :: String -> Bool
validComEmail email =
containsAtSign email && endsWithCom email
  where
    containsAtSign = (\e -> "@" `isInfixOf` e) :: Func
    endsWithCom = (\e -> ".com" `isSuffixOf` e) :: Func
```

Втім, на практиці вказівка типу для лямбда-абстракції зустрічається вкрай рідко, бо не є потрібною.

Віднині, познайомившись з ЛФ, ми будемо періодично їх використовувати.

І наостанок, питання. Пам'ятаєте тип функції `mul`?

```
mul :: a -> a -> a
```

Що це за буква `a`? По-перше, ми не зустрічали такий тип раніше, а по-друге, хіба ім'я типу в Haskell не зобов'язане починатися з великої літери? Зобов'язане. А вся справа в тому, що літера `a` в даному випадку — це не зовсім ім'я типу. А от що це таке, ми дізнаємося в одному з наступних розділів.

Для допитливих

А чому, власне, лямбда? Чому Черч вибрав саме цю грецьку букву? За однією з версій, це сталося випадково.

Йшли 30-ті роки минулого століття, комп'ютерів не було, і всі наукові роботи набиралися на друкарських машинках. У первісному варіанті, щоб виділяти ім'я аргументу ЛФ, Черч ставив над ім'ям аргументом символ, схожий на ˆ. Але коли він здавав роботу набірнику, то згадав, що друкарська машинка не зможе відтворити такий символ над буквою. Тоді він виніс цей «дашок» перед ім'я аргументу, і вийшло щось на кшталт:

```
^x . x * 10
```

А набірник, побачивши такий символ, використав велику грецьку букву Λ:

```
Λx . x * 10
```

Ось так і вийшло лямбда-числення.

Розділ 14

Композиція функцій

Цей розділ розповідає про те, як об'єднувати функції в ланцюжки, а також про те, як позбутися круглих дужок.

Дужкам — бій!

Так, я не люблю круглі дужки. Вони роблять код візуально надлишковим, до того ж потрібно стежити за симетрією відкриваючих і закриваючих дужок. Згадаймо приклад з розділу про кортежі:

```
main :: IO ()
main =
  putStrLn (patientEmail ( "63ab89d"
           ^             , "John Smith"
                         , "johnsm@gmail.com"
                         , 59
                         ))
                          ^
```

З дужками кортежу ми нічого зробити не можемо, адже вони є синтаксичною частиною кортежу. А ось дужки навколо застосування функції `patientEmail` мені абсолютно не подобаються. На щастя, ми можемо позбутися від них. Але перш ніж викорінювати дужки, подумаємо ось про що.

Якщо застосування функції є виразом, чи можемо ми якось компонувати їх одне з одним? Звичайно можемо, ми вже робили це багато разів, згадайте:

```
main :: IO ()
main = putStrLn (checkLocalhost "173.194.22.100")
```

Тут компонуються дві функції, `putStrLn` і `checkLocalhost`, тому що тип виразу на виході функції `checkLocalhost` збігається з типом виразу на вході функції `putStrLn`. Схематично це можна зобразити так:

```
          +--------------+            +--------+
String -> |checkLocalhost|-> String ->|putStrLn|-> ...
          +--------------+            +--------+

IP-адреса                    повідомлення          текст
                             про цю                в нашому
                             IP-адресу             терміналі
```

Виходить такий собі конвеєр: на вході рядок з IP-адресою, на виході — повідомлення у нашому терміналі. Існує інший спосіб поєднання двох функцій в одну.

Композиція та застосування

Погляньте:

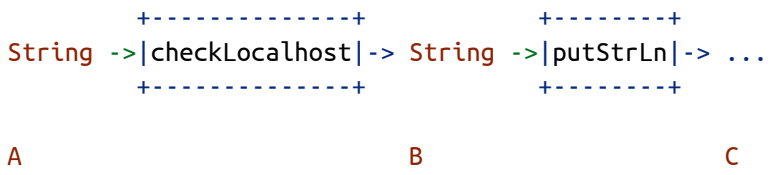
```
main :: IO ()
main = putStrLn . checkLocalhost $ "173.194.22.100"
```

Незвично? Перед нами два нових стандартних оператора, які позбавляють нас від зайвих дужок і роблять наш код простішим. Оператор `.` — це оператор композиції функцій (англ. function composition), а оператор `$` — це оператор застосування (англ. application operator). Ці оператори часто використовують спільно один з одним. І відтепер ми будемо використовувати їх чи не в кожному розділі.

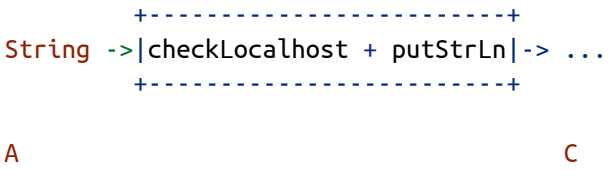
Оператор композиції об'єднує дві функції в одну (або компонує їх, англ. compose). Коли ми пишемо:

```
putStrLn . checkLocalhost
```

відбувається маленька «магія»: дві функції об'єднуються в нову функцію. Згадаймо наш конвеєр:



Якщо нам потрібно потрапити з точки `A` точку `C`, чи можна зробити це відразу? Можна, і в цьому полягає суть композиції: ми беремо дві функції і об'єднуємо їх в третю функцію. Раз `checkLocalhost` приводить нас з точки `A` точку `B`, а функція `putStrLn` — з точки `B` в `C`, тоді композиція цих двох функцій буде представляти собою функцію, що приводить нас відразу з точки `A` в точку `C`:



В даному випадку знак + не відноситься до конкретного оператора, я лише показую факт «об'єднання» двох функцій в третю. Тепер нам зрозуміло, чому в типі функції як роздільник використовується стрілка:

```
checkLocalhost :: String -> String
```

у нашому прикладі це:

```
checkLocalhost :: A -> B
```

Вона показує наш рух з точки `A` точку `B`. Тому часто говорять про «функції з `A` в `B`». Так, про функцію `checkLocalhost` можна сказати як про «функцію з `String` в `String`».

А оператор застосування працює ще простіше. Без нього код був би таким:

```
main :: IO ()
main =
  (putStrLn . checkLocalhost) "173.194.22.100"

  об'єднана функція           аргумент
```

Але ми хотіли позбутися круглих дужок, а тут вони знову. Ось для цього і потрібен оператор застосування. Його схема проста:

```
FUNCTION $ ARGUMENT
оця        застосовується до
функція    цього аргументу
```

Для нашої об'єднаної функції це виглядає так:

```
main :: IO ()
main =
  putStrLn . checkLocalhost $              "173.194.22.100"

  об'єднана функція          застосовується
                             до             цього аргументу
```

Тепер вийшов справжній конвеєр: праворуч у нього «заїжджає» рядок і рухається «крізь» функції, а зліва «виїжджає» результат:

```
main = putStrLn . checkLocalhost $  "173.194.22.100"

     <-         <-               <- аргумент
```

Щоб було легше читати композицію, замість оператора . подумки підставляємо фразу «застосовується після»:

```
putStrLn .              checkLocalhost

ця       застосовується цієї
функція  після         функції
```

Тобто композиція правоасоціативна (англ. right-associative): спочатку застосовується функція праворуч, а потім — ліворуч.

Ще одне зауваження про оператор застосування функції. Він дуже гнучкий, і ми можемо написати так:

```
main = putStrLn . checkLocalhost $ ”173.194.22.100”

       об’єднана функція          |_ її аргумент_|
```

а можемо і так:

```
main = putStrLn $ checkLocalhost ”173.194.22.100”

       звичайна   |________ її аргумент ________|
       функція
```

Ці дві форми, як ви вже зрозуміли, еквівалентні. Я показую це для того, щоб знову і знову продемонструвати вам, наскільки гнучко працювати з даними і функціями в Haskell.

Довгі ланцюжки

Краса композиції в тому, що компонувати ми можемо стільки функцій, скільки нам заманеться:

```
logWarn :: String -> String
logWarn rawMessage =
  warning . correctSpaces . asciiOnly $ rawMessage

main :: IO ()
main = putStrLn $
  logWarn ”Province ’Gia Vi□n’ isn’t on the map! ”
```

Функція `logWarn` готує переданий їй рядок для запису в журнал. Функція `asciiOnly` готує рядок висновку в нелокалізованому терміналі (так, у 2016 році такі все ще є), функція `correctSpaces` прибирає по-

двійні пробіли, а функція `warning` робить рядок попередженням (наприклад, додає рядок "`WARNING: `" на початку повідомлення). При запуску цієї програми ми побачимо:

```
WARNING: Province 'Gia Vi?n' isn't on the map!
```

Тут ми об'єднали в «функціональний конвеєр» вже три функції, без будь-яких дужок. Ось як це вийшло:

```
warning . correctSpaces . asciiOnly $ rawMessage

          ============= ^ =========

          |__  перша композиція __|

+++++++ ^ ++++++++++++++++++++++++

| ________ друга композиція ________|
                                      аргумент
```

Перша композиція об'єднує дві прості функції, `correctSpaces` і `asciiOnly`. Друга об'єднує теж дві функції, просту `warning` і об'єднану, яка є результатом першої композиції.

Більше того, визначення функції `logWarn` можна зробити ще більш простим:

```
logWarn :: String -> String
  logWarn = warning . correctSpaces . asciiOnly
```

Стривайте, але де ж ім'я аргументу? А його більше немає, воно нам не потрібно. Адже ми знаємо, що застосування функції можна легко замінити внутрішнім виразом функції. А раз так, вираз `logWarn` може бути замінено на вираз `warning . correctSpaces . asciiOnly`. Давайте це зробимо:

```
logWarn ”Province ’Gia Vi□n’ isn’t on the map! ”=

(warning
 . correctSpaces
 . asciiOnly) ”Province ’Gia Vi□n’ isn’t on the map! ”=

warning
. correctSpaces
. asciiOnly $ ”Province ’Gia Vi□n’ isn’t on the map! ”
```

І все працює! У світі Haskell прийнято саме так: якщо щось може бути спрощено — ми спрощуємо.

Заради справедливості слід зауважити, що не всі Haskell-розробники люблять позбавлятися від круглих дужок, деякі вважають за краще використовувати саме їх. Що ж, це питання лише стилю та звичок.

Як працює композиція

Якщо раптом ви подумали, що оператор композиції унікальний і вбудований в Haskell — поспішаю вас розчарувати. Ніякої магії, все гранично просто. Цей стандартний оператор визначений так само, як і будь-яка інша функція. Ось його визначення:

```
(.) f g = \x -> f (g x)
```

Опа! Та тут і справді немає нічого особливого. Оператор композиції застосовується до двох функцій. Стоп, скажете ви, як це? Застосовується до функцій? Так, саме так. Адже ми вже з’ясували, що функціями можна оперувати як даними. А раз так, що нам заважає передати функцію як аргумент для іншої функції? Що нам заважає

повернути функцію з іншої функції? Нічого.

Оператор композиції отримує на вхід дві функції, а потім лише дає нам ЛФ, всередині якої відбувається звичайний послідовний виклик цих двох функцій через дужки. І ніякої магії:

```
(.)    f       g       = \x -> f (g x)

беремо цю      цю        і повертаємо
       функцію функцію   ЛФ, всередині
                         якої
                         викликаємо їх
```

Підставимо наші функції:

```
(.) putStrLn checkLocalhost = \x -> putStrLn (checkLocalhost x)
```

Ось так і відбувається «об'єднання» двох функцій: ми просто повертаємо ЛФ від одного аргументу, всередині якої правоасоціативно викликаємо обидві функції. Аргументом в даному випадку є той самий рядок з IP-адресою:

```
(\x -> putStrLn (checkLocalhost x)) ”173.194.22.100” =

putStrLn (checkLocalhost ”173.194.22.100”))
```

Але якщо я вас ще не переконав, давайте визначимо власний оператор композиції функцій! Пам'ятаєте, я казав вам, що ASCII-символи можна гнучко поєднувати в оператори? Давайте візьмемо плюс зі стрілками, він чимось схожий на об'єднання. Пишемо:

```
-- Наш власний оператор композиції.
(<+>) f g = \x -> f (g x)

...

main :: IO ()
main = putStrLn <+> checkLocalhost $ "173.194.22.100"
```

Виглядає незвично, але працювати буде так, як і очікується: ми визначили власний оператор `<+>` з тим же функціоналом, що і стандартний оператор композиції. Тому можна написати ще простіше:

```
(<+>) f g = f . g
```

Ми кажемо: «Нехай оператор `<+>` буде еквівалентний стандартному оператору композиції функцій.». І так воно й буде. А можна — не повірите ще простіше:

```
f <+> g = f . g
```

І це буде працювати! Раз оператор призначений для інфіксного застосування, то ми, визначаючи його, можемо відразу вказати його в інфіксній формі:

```
f <+> g =      f . g

        нехай
такий
вираз
        буде
        рівним
               такому
               виразу
```

Тепер ми бачимо, що в композиції функцій немає нічого надприродного. На цьому я наголошую протягом всієї книги: Haskell не має ніякої магії, він логічний і послідовний.

Розділ 15

Функції Вищого Порядку

ФВП, або Функції Вищого Порядку (англ. HOF, Higher Order Functions) — важлива концепція в Haskell, з якою, однак, ми вже знайомі. Як ми дізналися з попередніх розділів, функціями можна оперувати як значеннями. Так от, функції, які оперують іншими функціями як аргументами і/або як результуючим вираженням, мають назву функцій вищого порядку.

Так, оператор композиції функцій є ФВП, тому що він, по-перше, приймає функції в якості аргументів, а по-друге, повертає іншу функцію (у вигляді ЛФ) як результат свого застосування. Використання функцій в якості аргументів — надзвичайно поширена практика в Haskell.

Відображення

Розглянемо функцію `map`. Ця стандартна функція використовується для відображення (англ. mapping) функції на елементи списку. Нехай вас не бентежить такий термін: відображення функції на елемент фактично означає її застосування до елемента.

Ось оголошення функції `map`:

```
map :: (a -> b) -> [a] -> [b]
```

Ось знову ці маленькі літери! Пам'ятаєте, я обіцяв розповісти про них? Розповідаю: малою літерою прийнято іменувати поліморфний (англ. polymorphic) тип. Поліморфізм — це різноманітність, багатоформенність. В даному випадку мова йде не про зазначення конкретного типу, а про «типову заглушку». Ми говоримо: «Функція `map` застосовується до функції з якогось типу `a` в якийсь тип `b` і до списку типу `[a]`, а результат її роботи — це інший список типу `[b]`». Типовими заглушками я назвав їх тому, що на їхнє місце стають конкретні типи, що робить функцію `map` дуже гнучкою. Наприклад:

```
import Data.Char

toUpperCase :: String -> String
toUpperCase str = map toUpper str

main :: IO ()
main = putStrLn . toUpperCase $ "haskell.org"
```

Результатом роботи цієї програми буде рядок:

```
HASKELL.ORG
```

Функція `map` застосовується до двох аргументів: до функції `toUpper` і рядка `str`. Функція `toUpper` зі стандартного модуля `Data.Char` переводить символ типу `Char` верхній регістр:

```
toUpper 'a' = 'A'
```

Ось її оголошення:

```
toUpper :: Char -> Char
```

Функція `Char` в `Char` виступає першим аргументом функції `map`, підставимо сигнатуру:

```
map :: (a -> b) -> [a] -> [b]
       (Char -> Char)
```

Ага, вже тепліше! Ми зробили два нових відкриття: по-перше, заглушки `a` і `b` можуть бути зайняті одним і тим самим конкретним типом, а по-друге, сигнатура дозволяє нам тут же зрозуміти інші типи. Підставимо їх:

```
map :: (a -> b)       -> [a]    -> [b]
       (Char -> Char)    [Char]    [Char]

        ____              ____

              ____                ____
```

А тепер згадаємо про природу типу `String`:

```
map :: (a -> b)       -> [a]    -> [b]
       (Char -> Char)    String    String
```

Все стало на свої місця. Функція `map` в даному випадку бере функцію `toUpper` і біжить по списку, послідовно застосовуючи цю функцію до його елементів:

```
map toUpper ['h','a','s','k','e','l','l','.','o','r','g']
```

Так, на першому кроці функція `toUpper` буде застосована до елементу `h`, на другому — до елементу `a`, і так далі до останнього елемента

g. Коли функція map біжить по цьому списку, результат застосування функції toUpper до його елементів стає елементами другого списку, який і буде в кінцевому підсумку повернуто. Так, результатом першого кроку буде елемент H, результатом другого — елемент A, а результатом останнього — елемент G. Схема така:

```
map toUpper [ 'h' >> [ 'H'
            , 'a' >> , 'A'
            , 's' >> , 'S'
            , 'k' >> , 'K'
            , 'e' >> , 'E'
            , 'l' >> , 'L'
            , 'l' >> , 'L'
            , '.' >> , '.'
            , 'o' >> , 'O'
            , 'r' >> , 'R'
            , 'g' >> , 'G'
            ]        ]
```

Ось і виходить:

```
map toUpper "haskell.org" = "HASKELL.ORG"
```

Робота функції map виглядає як зміна списку, однак, маючи на увазі незмінність останнього, насправді формується новий список. Що найцікавіше, функція toUpper перебуває в повному незнанні про те, що нею в кінцевому підсумку змінюють регістр цілого рядка, вона знає лише про окремі символи рядка. Тобто функція, що є аргументом функції map, нічого не знає про функцію map, і це дуже добре! Чим менше функції знають одна про одну, тим простіше і надійніше використовувати їх одну з одною.

Розглянемо інший приклад, коли типові заглушки a і b заміщуються різними типами:

```
toStr :: [Double] -> [String]
toStr numbers = show map numbers

main :: IO ()
main = print . toStr $ [1.2, 1,4, 1.6]
```

Функція `toStr` працює вже зі списками різних типів: на вході список чисел з плаваючою крапкою, на виході список рядків. При запуску цієї програми ми побачимо наступне:

```
[”1.2”,”1.0”,”4.0”,”1.6”]
```

Вже знайома нам стандартна функція `show` перетворює свій єдиний аргумент в рядковий вигляд:

```
show 1.2 = ”1.2”
```

В даному випадку, оскільки ми працюємо з числами типу `Double`, тип функції `show` такий:

```
show :: Double -> String
```

Підставимо в сигнатуру функції `map`:

```
map :: (a -> b)           -> [a]      -> [b]
       (Double -> String)    [Double]    [String]

         ______                 ______

                ======                  ======
```

Саме так, як у нас і є:

```
show map [1.2, 1,4, 1.6] = [”1.2”,”1.0”,”4.0”,”1.6”]
```

Функція `map` застосовує функцію `show` до чисел з першого списку, на виході отримуємо другий список, вже з рядками. Як і у випадку з `toUpper` функція `show` нічого не підозрює про те, що нею оперували в якості аргументу функції `map`.

Зрозуміло, в якості аргументу функції `map` ми можемо використовувати наші власні функції:

```
ten :: [Double] -> [Double]
ten = map (\n -> n * 10)

main :: IO ()
main = print . ten $ [1.2, 1,4, 1.6]
```

Результат роботи:

```
[12.0,10.0,40.0,16.0]
```

Ми передали функції `map` нашу власну ЛФ, яка множить свій єдиний аргумент на `10`. Зверніть увагу, ми знову використовували коротку форму визначення функції `ten`, опустивши ім’я аргументу. Розкриємо детальніше:

```
main = print .          ten $ [1.2, 1,4, 1.6] =
                  _____/  \_____
                 /              \
                /                \
main = print . map (\n -> n * 10) $ [1.2, 1,4, 1.6]
```

Ви запитаєте, як же вийшло, що оператор застосування розташований між двома аргументами функції `map`? Хіба він не призначений

для застосування функції до єдиного аргументу? Абсолютно вірно. Прийшов час відкрити ще один секрет Haskell.

Часткове застосування

Функція `map` очікує на два аргументи, це відображено в її типі. Але що буде, якщо застосувати її не до двох аргументів, а лише до одного? В цьому випадку відбудеться ще одне «магічне» перетворення, яке називається частковим застосуванням (англ. partial application) функції. Частковим називають таке застосування, коли аргументів менше ніж очікується.

Згадаймо скорочене визначення функції `ten`:

```
ten = map (\n -> n * 10)

          перший          а де ж
          аргумент        другий??
          є
```

Функція `map` отримала лише перший аргумент, а де ж другий? Другий, як ми вже знаємо, вона одержить вже потім, після того, як ми підставимо цей вираз на місце функції `ten`. Але що ж відбувається з функцією `map` до цього? А до цього з нею відбувається часткове застосування. Зрозуміло, що вона ще не може виконати свою роботу, тому, будучи застосованою лише до одного аргументу, вона повертає ЛФ! Зіставимо з типом функції `map`, і все стане на свої місця:

```
map :: (a -> b) -> [a] ->          [b]

map (\n -> n * 10)

    тільки перший
    аргумент

|    частково    |
|   застосована  |
|______ map _____| аргумент        відповідь
                  частково
                  застосованої
                  функції map

                  [1.2, 1,4, 1.6]
```

Тип ЛФ, поверненої після застосування `map` до першого аргументу — `[a] -> [b]`. Це «типовий хвіст», що залишився від повного типу функції `map`:

```
map :: (a -> b) -> [a]  ->  [b]

        голова     |_ хвіст __|
```

Оскільки голова у вигляді першого аргументу типу `(a -> b)` вже є, залишилося отримати другий аргумент. Тому ЛФ, породжена частковим застосуванням, очікує єдиний аргумент, яким і буде той самий другий, а саме список`[1.2, 1,4, 1.6]`.

Зіставимо тип функції `ten` з типом `map`, щоб зрозуміти, де наш хвіст:

```
ten ::              [Double] -> [Double]

map :: (a -> b) -> [a]       -> [b]

        голова      |_____ хвіст ______|
```

Ось чому ми можемо використовувати коротку форму для визначення функції `ten`: вона вже є нашим хвостом!

Розглянемо ще один приклад часткового застосування, щоб закріпити наше розуміння:

```
replace :: String -> String -> String -> String
```

Це оголошення функції `replace`, яка приймає три рядки: перша містить те, що шукаємо, друга містить те, на що замінюємо, а в третій лежить те, де шукаємо. Наприклад:

```
replace "http"
        "https"
        "http://google.com" = "https://google.com"
```

Визначення функції `replace` нас зараз не цікавить, розглянемо покрокове застосування:

```
main :: IO ()
main = result putStrLn
  where
    first = replace "http"
    second = first "https"
    result = second "http://google.com"
```

Тип виразу `first` — `String -> String -> String`, він став результатом

часткового застосування функції `replace` до першого аргументу, рядка "`http`". Тип виразу `second` — `String -> String`, він став результатом вторинного часткового застосування функції `first` до другого аргументу, рядка "`https`". І нарешті, застосувавши функцію `second` до третього аргументу, рядка "`http://google.com`", ми нарешті отримуємо кінцевий результат, асоційований з виразом `result`.

З цього ми робимо цікаве відкриття:

> Функція від декількох аргументів може бути розкладена на послідовність застосувань тимчасових функцій від одного аргументу кожна.

Тому ми і змогли підставити частково застосовану `map` на місце виразу `ten`. Використовуємо круглі дужки, щоб ясніше показати, що є що:

```
main = print . (map (\n -> n * 10)) $ [1.2, 1,4, 1.6]

               | частково          |
               | застосована map   |

        |      композиція функції  |
        |      print і частково    |
        |_____ застосованої map ___|
                                        аргумент для
                                        композиції
```

Гнучко, чи не так? Тепер ми знайомі з частковим застосуванням функції.

Композиція для відображення

Повернемося до функції `map`. Якщо ми можемо передати їй якусь функцію для роботи з елементами списку, значить ми можемо передати їй і композицію двох або більше функцій. Наприклад:

```
import Data.Char

pretty :: [String] -> [String]
pretty = map (stars . big)
  where
    big = map toUpper
    stars = \s -> "* "++ s ++ " *"

main :: IO ()
main = print . pretty $ ["haskell", "lisp", "coq"]
```

Ми хочемо прикрасити імена трьох мов програмування. Для цього ми пробігаємося по списку композицією двох функцій, `big` і `stars`. Функція `big` переводить терміни у верхній регістр, а функція `stars` прикрашає ім'я двома зірочками на початку і в кінці. В результаті маємо:

```
["* HASKELL *","* LISP *","* COQ *"]
```

Пройтися по списку композицією `stars . big` рівнозначно тому, якби ми пройшлися спочатку функцією `big`, а потім функцією `stars`. При цьому, як ми вже знаємо, обидві ці функції нічого не знають ні про те, що їх скомпонували, ні про те, що цю композицію передали функції `map`.

Ну що ж, тепер ми знаємо про функції `map`, і в наступних розділах ми побачимо безліч інших ФВП. Відтепер вони будуть нашими постійними супутниками.

Розділ 16

Hackage та бібліотеки

Раніше я вже згадував про бібліотеки, прийшов час познайомитися з ними ближче, адже в наступних розділах ми будемо використовувати їх постійно.

Бібліотеки великі та маленькі

За роки існування Haskell розробники з усього світу створили безліч бібліотек. Бібліотеки рятують нас від необхідності знову і знову писати те, що вже написано до нас. Для будь-якої живої мови програмування написано безліч бібліотек. У світі Haskell їх, звичайно, не така хмара, як для тієї ж Java, але досить багато: стабільних є не менше двох тисяч, багато з яких дуже якісні та вже багаторазово випробувані в серйозних проектах.

З модулями — файлами, що містять Haskell-код, — ми вже знайомі, вони є основними цеглинами будь-якого Haskell-проекту. Бібліотека, будучи Haskell-проектом, теж складається з модулів (не важливо, з одного або з сотень). Тому використання бібліотеки зводиться

до використання модулів, які в неї входять. І ми вже неодноразово робили це в попередніх розділах.

Згадаймо приклад з розділу про ФВП:

```
import Data.Char

toUpperCase :: String -> String
toUpperCase str = map toUpper str

main :: IO ()
main = putStrLn . toUpperCase $ "haskell.org"
```

Функція `toUpper` визначена в модулі `Data.Char`, який, у свою чергу, знаходиться в стандартній бібліотеці. Бібліотек є безліч, але стандартна лише одна. Вона містить основні інструменти, які вживаються найбільш широко. Але, перш ніж продовжити, поставимо собі важливе питання: «Де знаходяться всі ці бібліотеки?» Вони знаходяться у різних місцях, але головне з них — Hackage.

Hackage

Hackage — це центральний репозиторій Haskell-бібліотек, або, як прийнято у нас називати, пакетів (англ. package). Назва репозиторію походить від злиття слів `Haskell` і `package`. Hackage існує з 2008 року і знаходиться тут. Раніше згадана стандартна бібліотека теж живе в Hackage і називається вона `base`. Кожній бібліотеці виділена своя сторінка.

Кожен з Hackage-пакетів живе за адресою, сформованою за незмінною схемою: `http://hackage.haskell.org/package/IM'ЯПАКЕТУ`. Так, дім стандартної бібліотеки — `http://hackage.haskell.org/package/base`. Hackage — відкритий репозиторій: будь-який розробник може додати туди свої пакети.

Стандартна бібліотека включає в себе більше сотні модулів, але є серед них найвідоміший, що носить ім'я `Prelude`. Цей модуль за замовчуванням завжди з нами: весь його вміст автоматично імпортується в усі модулі нашого проекту. Наприклад, вже відомі нам `map` або оператори конкатенації списків живуть в модулі `Prelude`, тому доступні нам завжди. Крім них (і багатьох-багатьох десятків інших функцій) в `Prelude` розташовуються функції для роботи з введенням-виведенням, такі як наші знайомі `putStrLn` і `print`.

Hackage досить великий, тому шукати пакети можна двома способами. Перший — на єдиній сторінці всіх пакетів. Тут перераховані всі пакети, а для нашої зручності вони розташовані за тематичними категоріями.

Другий спосіб — через спеціальний пошуковик, яких є два:

1. Hoogle
2. Hayoo!

Ці пошуковики скрупульозно переглядають нутрощі Hackage, і ви будете часто ними користуватися. Особисто я віддаю перевагу Hayoo!. Користуємося ним як звичайним пошуковиком: наприклад, ми знаємо ім'я функції, а в якому пакеті/модулі вона міститься — забули. Вводимо в пошук — отримуємо результати.

Щоб скористатися пакетом в нашому проекті, потрібно для початку включити його в наш проект. Для прикладу розглянемо пакет `text`, призначений для роботи з текстом. Він нам знадобиться, тому включимо його в наш проект негайно.

Відкриваємо файл збірки проекту `real.cabal`, знаходимо секцію `executable real-exe` у полі `build-depends` через кому дописуємо ім'я пакета:

```
build-depends: base -- Вже тут!
             , real
             , text -- А це новий пакет.
```

Файл з розширенням `.cabal` — це обов'язковий файл збірки Haskell-проекту. Він містить головні інструкції, що стосуються збирання проекту. З синтаксисом складального файлу ми будемо поступово знайомитися в наступних розділах.

Як бачите, пакет `base` вже тут. Включивши пакет `text` в секцію `build-depends`, ми оголосили тим самим, що наш проект відтепер залежить від цього пакету. Тепер, перебуваючи в корені проекту, виконуємо вже знайому нам команду:

```
$ stack install
```

Пам'ятаєте, коли ми вперше налаштовували проект, я згадав, що утиліта `stack` вміє ще й бібліотеки встановлювати? Вона побачить нову залежність нашого проекту і встановить сам пакет `text`, так і всі ті пакети, від яких, в свою чергу, залежить пакет `text`. Після збірки ми можемо імпортувати модулі з цього пакету в наші модулі. І тепер прийшла пора дізнатися, як це можна робити.

Ієрархія імен

Коли ми пишемо:

```
import Data.Char
```

в імені модуля відображена ієрархія пакета. `Data.Char` означає, що всередині пакету `base` каталог `Data`, всередині якого живе файл `Char.hs`, відкривши який, ми побачимо:

```
module Data.Char
...
```

Таким чином, точка в імені модуля відображає файлову ієрархію всередині даного пакету. Можете сприймати цю точку як слеш в Unix-шляху. Є пакети зі значно більш довгими іменами, наприклад:

```
module GHC.IO.Encoding.UTF8
```

Відповідно, імена наших власних модулів теж відображають місце, в якому вони знаходяться. Так, один з модулів в моєму робочому проекті носить назву `Common.Performers.Click`. Це означає, що знайти цей модуль можна тут: `src/Common/Performers/Click.hs`.

Обличчя

Повернемося до нашого прикладу:

```
import Data.Char
```

Імпорт модуля `Data.Char` робить доступним для нас все те, що включено в інтерфейс цього модуля. Відкриємо наш власний модуль `Lib`:

```
module Lib
    ( someFunc
    ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

Ім'я функції `someFunc` згадано в інтерфейсі модуля, а саме між круглими дужками, що йдуть після імені модуля. Трохи переформатуємо дужки:

```
module Lib (
  someFunc
) where
```

Зараз тільки функція `someFunc` доступна всім імпортерам даного модуля. Якщо ж ми визначимо в цьому модулі іншу функцію `anotherFunc`:

```
module Lib (
  someFunc
) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"

anotherFunc :: String -> String
anotherFunc s = s ++ "!"
```

вона залишиться невидимою для зовнішнього світу, тому що її ім'я не згадано в інтерфейсі модуля. І якщо в модулі `Main` ми напишемо так:

```
module Main

import Lib

main :: IO ()
main = putStrLn . anotherFunc $ "Hi"
```

компілятор справедливо насвариться, мовляв, не знаю функцію `anotherFunc`. Якщо ж ми додамо її в інтерфейс модуля `Lib`:

```
module Lib (
  someFunc,
  anotherFunc
) where
```

тоді функція `anotherFunc` теж стане видимою всьому світу. Інтерфейс дозволяє нам показувати оточуючим лише те, що ми хочемо їм показати, залишаючи службову структуру нашого модуля таємницею за сімома печатями.

Імпортуємо по-різному

У реальних проектах ми імпортуємо безліч модулів з різних пакетів. Іноді це є причиною конфліктів, з якими доводиться мати справу.

Згадаймо функцію `putStrLn`: вона існує не тільки у незримому модулі `Prelude`, але й в модулі `Data.Text.IO` з пакету `text`:

```
-- Тут теж є функція на ім'я putStrLn.
import Data.Text.IO

main :: IO ()
main = putStrLn ... -- І звідки ця функція?
```

При спробі зібрати такий код ми зіткнемося з помилкою:

```
Ambiguous occurrence 'putStrLn'
It could refer to either 'Prelude.putStrLn',
imported from 'Prelude' ...
or 'Data.Text.IO.putStrLn',
imported from 'Data.Text.IO' ...
```

Нам необхідно якось вказати, яку з функцій `putStrLn` ми маємо на увазі. Це можна зробити кількома способами.

Можна вказати приналежність функції до конкретного модуля. З повідомлення про помилку вже видно, як це можна зробити:

```
-- Тут теж є функція на ім'я putStrLn.
import Data.Text.IO

main :: IO ()
main = Data.Text.IO.putStrLn ... -- Сумнівів немає!
```

Тепер уже сумнівів не залишилося: використовувана нами `putStrLn` належить модулю `Data.Text.IO`, тому колізій немає.

Втім, чи не здається вам така форма занадто довгою? У згаданому раніше стандартному модулі `GHC.IO.Encoding.UTF8` функція `mkUTF8`, і уявіть собі:

```
import GHC.IO.Encoding.UTF8

main :: IO ()
main =
  let enc = GHC.IO.Encoding.UTF8.mkUTF8 ...
```

Занадто довго, потрібно вкоротити. Імпортуємо модуль під короткою назвою:

```
import    Data.Text.IO  as  TIO

включити  цей модуль    як  це

main :: IO ()
main = TIO.putStrLn ...
```

Ось, так значно краще. Коротке ім'я може складатися навіть з однієї літери, але як і повне ім'я модуля, воно обов'язково повинно починатися з великої літери, тому:

```
import Data.Text.IO as tIO -- Помилка
import Data.Text.IO as i -- Теж помилка
import Data.Text.IO as I -- Порядок!
```

Іноді, для більшого порядку, використовують qualified-імпорт:

```
import qualified Data.Text.IO as TIO
```

Ключове слово `qualified` використовується для «строгого» включення модуля: в цьому випадку ми зобов'язані вказувати належність до нього. Наприклад:

```
import qualified Data.Text as T

main :: IO ()
main = T.justifyLeft ...
```

Навіть незважаючи на те, що функція `justifyLeft` є тільки в модулі `Data.Text` і ніяких колізій з `Prelude` немає, ми зобов'язані вказати, що ця функція саме з `Data.Text`. У великих модулях qualified-імпорт буває корисний: з одного боку, гарантовано не буде ніяких конфліктів, з іншого, ми відразу бачимо, звідки походить та чи інша фун-

кція.

Втім, деяким Haskell-програмістам будь-яка вказівка приналежності до модуля здається надмірною. Тому вони йдуть іншим шляхом: вибіркове включення/приховування. Наприклад:

```
import Data.Char
import Data.Text (pack) -- Тільки її!

main :: IO ()
main = putStrLn $ map toUpper "haskell.org"
```

Ми маємо на увазі стандартну функцію `map`, проте в модулі `Data.Text` теж міститься функція на ім'я `map`. На щастя, ніякої колізії не буде, адже ми імпортували не весь вміст модуля `Data.Text`, а лише одну його функцію `pack`:

```
import     Data.Text (pack)

імпортуємо звідси    тільки
                     це
```

Якщо ж ми хочемо імпортувати дві чи більше функції, перераховуємо їх через кому:

```
import Data.Text (pack, unpack)
```

Існує і протилежний шлях: замість вибіркового включення — вибіркове приховування. Уникнути колізії між функціями `putStrLn` можна було б і так:

```
import Data.Text.IO hiding (putStrLn)

main :: IO ()
main = putStrLn ... -- Сумнівів немає: Prelude.
```

Слово `hiding` дозволяє приховувати дещо з імпортованого модуля:

```
import     Data.Text.IO hiding (putStrLn)

імпортуємо всі звідси   крім   цього
```

Можна й кілька функцій приховати:

```
import Data.Text.IO hiding ( readFile
                           , writeFile
                           , appendFile
                           )
```

При бажанні можна приховати і з `Prelude`:

```
import Prelude hiding (putStrLn)
import Data.Text.IO

main :: IO ()
main = putStrLn ... -- Вона точно з Data.Text.IO.
```

Оформлення

Загальна рекомендація така — оформляйте так, щоби було легше читати. В реальному проекті в кожен з ваших модулів буде імпорту-

ватися досить багато всього. Ось шматочок з одного мого робочого модуля:

```
import qualified Test.WebDriver.Commands as WDC
import           Test.WebDriver.Exceptions
import qualified Data.Text as T
import           Data.Maybe (fromJust)
import           Control.Monad.IO.Class
import           Control.Monad.Catch
import           Control.Monad (void)
```

Як повні, так і короткі імена модулів вирівняні, такий код простіше читати і змінювати. Не всі програмісти погодяться з таким стилем, але спробуємо прибрати вирівнювання:

```
import qualified Test.WebDriver.Commands as WDC
import Test.WebDriver.Exceptions
import qualified Data.Text as T
import Data.Maybe (fromJust)
import Control.Monad.IO.Class
import Control.Монада.Catch
import Control.Монада (void)
```

Тепер код виглядає зім'ятим, його важче сприймати. Втім, вибір за вами.

Розділ 17

Рекурсія

> Щоб зрозуміти рекурсію потрібно спочатку зрозуміти рекурсію.

Цей старий жарт про рекурсію іноді лякає новачків, як у свій час налякав мене. Насправді в рекурсії немає нічого страшного, і в цьому розділі ми познайомимося з цим важливим механізмом.

Цикл

Дивно, але в Haskell немає вбудованих циклічних конструкцій, настільки звичних для інших мов. Ні тобі `for`, ні `while`. Однак обійтися без циклів в нашому коді ми не зможемо. Як же нам їх організовувати?

На щастя, найчастіше нам це й не потрібно. Згадаймо нашу знайому, функцію `map`:

```
map toUpper someList
```

Ну і чим же не цикл? На тому ж C це виглядало би так:

```
int length = ...
for(int i = 0; i < length; ++i) {
    char result = toUpper(someList[i]);
...
}
```

Функції, на зразок `map`, в переважній більшості випадків позбавляють нас від написання явних циклічних конструкцій, і це радує. Однак зрідка нам все таки доведеться писати цикли явно. В Haskell, через відсутність `for`-конструкції, зробити це можна тільки одним способом — через рекурсію (англ. recursion).

Ідея рекурсії дуже проста:

> Якщо нам потрібно повторити обчислення, зроблене певною функцією, ми повинні застосувати цю функцію всередині себе самої. І вийде зациклення.

Поглянемо на визначення функції `map`:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

А тепер розберемо це цікаве визначення по кісточках.

Правда списку

Першим аргументом виступає певна функція, а другим — список, до елементів якого застосовується ця функція. Але що це за дивного вигляду конструкція в круглих дужках?

```
(x:xs)
```

Це — особливий зразок, який використовується для роботи зі списками. Щоб він став зрозумілим, я повинен розповісти вам правду про формування списку.

Як ми пам'ятаємо, формується список дуже просто:

```
[1, 2, 3] -- Список з трьох цілих чисел.
```

Проте насправді він формується дещо інакше. Звична нам конструкція в квадратних дужках є ніщо інше, як синтаксичний цукор (англ. syntactic sugar). Синтаксичним цукром називають якесь спрощення коду, що робить його солодшим, приємнішим для нас. Якщо ж ми приберемо цукор (або, як ще кажуть, «розцукруємо» код), то побачимо ось що:

```
1 : 2 : 3 : []
```

Саме так список з трьох цілих чисел формується насправді. Стандартний оператор : нам вже знайомий, ми зустрілися з ним у главі про списки:

```
newHost :        hosts

        цей
        оператор

бере
це
значення
                 і додає
                 його на початок
                 цього списку
```

Тобто список будується шляхом додавання елемента в його «голову», починаючи з порожнього списку:

```
1 : 2 : 3 : []

= 1 : 2 : [3]

= 1 : [2, 3]

= [1, 2, 3]
```

Починаючи з правого краю, ми спочатку застосовуємо оператор `:` до `3` і порожнього списку, в результаті чого отримуємо список з одним елементом `[3]`. Потім, застосовуючи другий оператор `:` до `2` і до щойно отриманого списку `[3]`, ми отримуємо новий список `[2, 3]`. Врешті, знову застосувавши оператор `:` до `1` і до списку `[2, 3]`, ми отримуємо підсумковий список`[1, 2, 3]`. Ось чому так зручно оперувати «головою» і «хвостом» списку. І саме тому був створений особливий зразок для патерн-матчінгової роботи зі списком:

```
(head : tail)
```

У даному випадку слова `head` та `tail` не належать до стандартних функцій, я лише показую призначення елементів даного зразка. Ось живий приклад:

```
main :: IO ()
main = first print
  where
    (first:others) = [”He”, ”Li”, ”Be”]

     _____              ____

           ======           ==========
```

Оскільки ми точно знаємо, що справа у нас список, зліва ми пишемо зразок списку, в якому `first` асоційований з першим елементом, з «головою», а шаблон `others` — з рештою елементів, з «хвостом».

Але ви запитаєте, навіщо нам це потрібно? Якщо вже ми так хочемо працювати зі списком через патерн матчінг, можна скористатися явним зразком:

```
main :: IO ()
main = first print
where
[first, second, third] = [”He”, ”Li”, ”Be”]

 _____                      ____

       ======                  ====

              +++++                 ++++
```

Все вірно, однак зразок з круглими дужками надзвичайно зручний саме для рекурсивної роботи зі списком, і ось чому. Згадаймо ви-

значення функції `map`:

```
map f (x:xs) = f x : map f xs

       _         _

         ==                ==
```

Підставимо реальні значення на основі прикладу про переведення символів рядка в верхній регістр:

```
map f (x:xs) = f x : map f xs

map toUpper "neon" = toUpper 'n' : map toUpper "eon"

             _                _

              ===                            ===
```

Ось тепер ми бачимо, яким чином функція `map` пробігає по всьому списку. Пройдемося по ітераціях, щоб все остаточно стало на свої місця. У нас же цикл, вірно? А де цикл — там ітерації.

На першій з них оператор `:` застосовується до виразів `toUpper 'n'` та `map toUpper "eon"`. Вираз зліва обчислюється й дає нам символ `H`:

```
toUpper 'n' : map toUpper "eon"

'H'         : map toUpper "eon"
```

Вираз праворуч містить застосування тієї ж функції `map`, тобто ми входимо в цикл, у другу ітерацію:

```
map toUpper "eon" = toUpper 'e' : map toUpper "on"
```

Вираз зліва обчислюється й дає нам `E`:

```
toUpper 'e' : map toUpper "on"

'E'         : map toUpper "on"
```

Обчислюємо вираз праворуч — та входимо в наступну ітерацію:

```
map toUpper "on" = toUpper 'o' : map toUpper "n"
```

Вираз зліва дає нам `O`:

```
toUpper 'o' : map toUpper "n"

'O'         : map toUpper "n"
```

Справа знову застосування `map` — й наша остання ітерація:

```
map toUpper "n" = toUpper 'n' : map toUpper []
```

Вираз зліва дає нам `N`:

```
toUpper 'n' : map toUpper []

'N'         : map toUpper []
```

Ми витягнули зі списку останній з чотирьох символів, і список залишився порожнім. Що ж ми будемо робити далі? А далі ми згадуємо перший варіант визначення функції `map`:

```
map _ [] = []
```

Тут функція каже: «Як тільки я другим аргументом отримаю порожній список, я, ігноруючи перший аргумент, негайно поверну той самий порожній список». Тому вираз, який залишився на останній ітерації праворуч:

```
map toUpper []
```

підійде під даний випадок і просто дасть нам порожній список. Все готово, робота функції завершена. На кожній ітерації ми відкушуємо «голову» списку і передаємо її функції `toUpper`, «хвіст» же передаємо знову функції `map`. На четвертій ітерації зупиняємося на порожньому списку і повертаємо його. Поєднавши всі ітерації воєдино, отримуємо ось що:

```
'N' : 'E' : 'O' : 'N' : []
```

Впізнаєте? Це ж наш «розцукрований» список, що з'єднується воєдино:

```
['N', 'E', 'O', 'N']
```

Ось ми й прийшли до нашої рівності:

```
map toUpper "neon"

= map toUpper ['n', 'e', 'o', 'n']

= ['N', 'E', 'O', 'N']

= "NEON"
```

Туди й назад

Визначаючи рекурсивну функцію, важливо пам'ятати про те, що в ній, як правило, повинно бути як і зациклення, так і правило виходу з циклу:

```
map _ [] = [] -- Виходимо з циклу.
map f (x:xs) = f x : map f xs -- Зациклюємося,
                              -- застосовуючи саму себе.
```

Якщо б ми опустили перше визначення, компілятор завбачливо повідомив би нам про проблему:

```
Match Pattern(es) are non-exhaustive
```

Це абсолютно правильно: якщо на кожній ітерації ми зменшуємо список, то рано чи пізно список точно залишиться порожнім, а отже, ми зобов'язані пояснити, що ж робити в цьому випадку.

Для допитливих

Відкрию таємницю: рекурсивними в Haskell бувають не тільки функції, але й типи. Але про це в наступних розділах.

Розділ 18

Ледачість

Пам'ятаєте, в розділі про перші питання про Haskell я згадував, що ця мова є ледачою? Зараз ми нарешті дізнаємося про ледачі обчислення та познайомимося з їхніми світлою та темною сторонами.

Дві моделі обчислень

Як ми вже знаємо, Haskell-програма складається з виразів, а запуск програми це початок довгого ланцюжка обчислень. Згадаймо функцію `square`, яка підносить свій єдиний аргумент до квадрату:

```
main :: IO ()
main = print . square $ 4
```

Тут все просто: функція `square` застосовується до нередукованого виразу `4` і дає нам `16`. А якщо так:

```
main :: IO ()
main = print . square $ 2 + 2
```

Тепер функція `square` застосовується вже до редукованого виразу:

```
square  $              2 + 2

функція застосовується редукованого
        до             виразу
```

Як ви думаєте, що станеться раніше? Застосування оператора додавання чи застосування функції `square`? Питання хитре, адже правильної відповіді на нього немає, оскільки існує дві моделі обчислення аргументів, а саме енергійна (англ. eager) і ледача (англ. lazy).

При енергійній моделі (яку також називають «жадібною» чи «строгою») вираз, що є аргументом функції, буде обчислено ще до того, як він потрапить у тіло функції. На тлі визначення функції `square` буде зрозуміліше:

```
square     x   = x * x
          /   \
square $ 2 + 2
          \   /
            4   = 4 * 4 = 16
```

Тобто бачимо вираз `2 + 2`, жадібно на нього накидуємося, повністю обчислюємо, а вже потім результат цього обчислення передаємо в функцію `square`.

При ледачій моделі все навпаки: вираз, що є аргументом функції, передається у функцію прямо так, без обчислення. Зобразити це

можна наступним чином:

```
square     x =      x    *    x
         /   \    /   \     /   \
square $ 2 + 2 = (2 + 2) * (2 + 2) = 16
```

Але яка різниця, запитаєте ви? Все одно в підсумку отримаємо `16`, хоч там додаємо, хоч тут. Так і є: модель обчислення не впливає на результат цього обчислення, але вона впливає на дорогу до цього результату.

Жадібна модель знайшла своє втілення практично у всіх сучасних мовах програмування. Напишемо на C:

```
#include <stdio.h>

int strange(int i) {
    return 22;
}

int main() {
    printf("%d\n", strange(2 / 0));
}
```

Функція `strange` дійсно дивна, адже вона ігнорує свій аргумент та просто повертає число `22`. Та все таки при запуску цієї програми ви гарантовано отримаєте помилку `Floating point exception`, бо компілятор мови C категорично не терпить ділення на нуль. А все тому, що мова C дотримується енергійної моделі обчислень: оператор ділення `2` на `0` буде викликаний ще до того, як ми увійдемо в тіло функції `strange`, тому програма перестане працювати.

Такий підхід прямолінійний і строгий: нам спочатку наказали поділити на нуль — поділимо, не замислюючись. Ледача ж модель дотримується іншого підходу. Погляньте на Haskell-варіант:

```
strange :: Int -> Int
strange i = 22

main :: IO ()
main = print . strange $ 2 `div` 0
```

Дивно, але при запуску цієї програми ми побачимо:

```
22
```

Втім, чому дивно? Функція `strange`, проігнорувавши свій аргумент, повернула нам значення `22`, яке, потрапивши на вхід функції `print`, потрапило в наш термінал. Але де ж помилка ділення `2` на `0`, запитаєте ви? Її немає.

Ледачий підхід цілком гармоніює зі своєю назвою: нам ліньки робити роботу одразу. Замість цього ми, подібно дитині, яку змусили прибрати розкидані по кімнаті іграшки, відкладаємо роботу до останнього. Ледача модель гарантує, що робота буде виконана лише тоді, коли результат цієї роботи комусь знадобиться. Якщо ж він нікому не знадобиться, тоді робота не буде виконана.

Функція `strange` ледача, а тому раціональна. Вона дивиться на свій аргумент `i`:

```
strange i = 22
```

і розуміє, що він ніде не використовується в її тілі. Значить, він не потрібен. А якщо так, то й обчисленим він не буде. До речі, якщо аргумент функції ігнорується, визначення прийнято писати з універсальним зразком:

```
strange _ = 22

        ^
        нам
        все
        одно
```

Так і виходить:

```
strange       _     = 22
          /       \
strange $ 2 `div` 0 = 22
```

Вираз, що містить ділення на нуль, потрапляє всередину функції, будучи ще необчисленим, але оскільки в тілі функції він ніде не використовується, він так і залишиться необчисленим. Девіз лінощів: якщо результат роботи нікому не потрібен — навіщо ж його робити? Ось чому фактичного ділення на нуль тут не відбудеться і програма не перестане працювати.

Зрозуміло, якщо б ми визначили функцію `square` інакше:

```
strange :: Int -> Int
strange i = i + 1
```

тоді інша справа: значення аргумента вже використовується в тілі функції, а значить обчислення аргумента неодмінно відбудеться:

```
strange       i     =      i      + 1
          /       \    /       \
strange $ 2 `div` 0 = (2 `div` 0) + 1
```

Оператору додавання потрібні значення обох його аргументів, в тому числі лівого, а отже ви отримаєте помилку ділення на нуль.

Якомога менше

Доки результат обчислення нікому не потрібен, воно не проводиться. Проте навіть тоді, коли результат комусь знадобився, обчислення відбувається не до кінця. Пам'ятаєте, вище я сказав, що при жадібній моделі обчислення вираз, що є аргументом, обчислюється «повністю»? А ось при ледачій моделі ми обчислюємо вираз лише настільки, наскільки це є необхідним. Так само як згадана раніше дитина, прибираючи іграшки в кімнаті, прибирає їх зовсім не до кінця, а лише до такої міри, щоб її не лаяли батьки.

З точки зору обчислення будь-який вираз в Haskell проходить через три стадії:

1. необчислений,
2. обчислений не до кінця,
3. обчислений до кінця.

Необчисленим називається такий вираз, який взагалі не чіпали. Згадаймо вищезазначене ділення на нуль:

```
2 `div` 0
```

Ми побачили, що програма не перестала працювати, і це каже нам про те, що ділення не відбулося. Тобто функція `div` так і не була застосована до своїх аргументів. Взагалі. Такий вислів називають thunk (можна перекласти як «задумка»). Тобто ми задумали застосувати функцію `div` до `2` і `0`, приготувалися зробити це — але в підсумку так і не зробили.

Обчисленим до кінця називають такий вираз, який обчислений до

своєї остаточної, нередукованої форми. Про такий вираз кажуть як про вираз в «нормальній формі» (англ. normal form).

А ось обчисленим не до кінця називають такий вираз, що почали обчислювати, але зробили це не до кінця, тобто не до нормальної форми, а до так званої «слабкої головної форми» (англ. Weak Head Normal Form, WHNF). Ви запитаєте, як же це можна обчислити вираз не до кінця? Розглянемо приклад:

```
main :: IO ()
main =
  let cx = 2 / 6.054 -- thunk
      nk = 4 * 12.003 -- thunk
      coeffs = [cx, nk] -- thunk
  in putStrLn "Nothing..."
```

В нас є два коефіцієнти, `cx` та `nk`, і ще список `coeffs`, в який ми помістили ці коефіцієнти. Але, як ми бачимо, у результаті ні ці коефіцієнти, ні цей список нам не знадобилися: ми просто вивели рядок і тихо вийшли. У цьому випадку жоден з цих виразів так і не було обчислено, вони залишились у вигляді thunk. Тобто оператор ділення так і не був застосований до `2` та `6.054`, оператор множення не торкнувся ні `4`, ні `12.003`, а список залишився лише в наших головах. Лінива стратегія раціональна: навіщо витрачати комп'ютерні ресурси на створення того, що в підсумку нікому не знадобиться?

Змінимо код:

```
main :: IO ()
main =
  let cx = 2 / 6.054 -- thunk
      nk = 4 * 12.003 -- thunk
      coeffs = [cx, nk] -- WHNF
  in print $ length coeffs
```

Ага, вже цікавіше. Цього разу нам захотілося дізнатися довжину

списку `coeffs`. У цьому випадку нам уже не обійтися без списку, інакше як же ми дізнаємося його довжину? Однак фокус в тому, що вираз `[cx, nk]` обчислюється не до кінця, а лише до тієї своєї форми, яка задовільнить функцію `length`.

Замислимося: функція `length` повертає кількість елементів списку, але яке їй діло до вмісту цих елементів? Зовсім ніякого. Тому в даному випадку список формується з thunk-ів:

```
coeffs = [thunk, thunk]
```

Першим елементом списку є thunk, асоційований з необчисленим виразом `2 / 6.054`, а другим елементом списку є thunk, асоційований з необчисленим виразом `4 * 12.003`. Фактично, список `coeffs` вийшов би не зовсім справжнім: він би був сформованим в пам'яті як коректний список, однак всередині обох його елементів — вакуум. Все таки навіть така його форма цілком підходить для функції `length`, яка й так прекрасно зрозуміє, що у списку є два елементи. Про такий список говорять як про вираз в слабкій головній формі.

Ще трохи змінимо код:

```
main :: IO ()
main =
  let cx = 2 / 6.054 -- thunk
      nk = 4 * 12.003 -- normal
      coeffs = [cx, nk] -- WHNF
  in print $ coeffs !! 1
```

Незвичний оператор `!!` дістає зі списку елемент за індексом, у даному випадку нас цікавить другий за порядком елемент. Тепер для нас вже недостатньо просто сформувати список, нам дійсно потрібен його другий елемент, інакше як би ми змогли вивести його на консоль? У цьому випадку вираз `4 * 12.003` буде обчислено до своєї

остаточної, нормальної форми, а результат цього обчислення стане другим елементом списку, ось так:

```
coeffs = [thunk, 48.012]
```

Однак перший елемент списку так і залишився непотрібним, тому вираз `2 / 6.054` залишається не більше ніж нашою задумкою. В цьому випадку список `coeffs` все одно залишається у слабкій головній формі, адже всередині першого елемента все ще вакуум.

А тепер напишемо так:

```
main :: IO ()
main =
  let cx = 2 / 6.054 -- normal
      nk = 4 * 12.003 -- normal
      coeffs = [cx, nk] -- normal
  in print coeffs
```

Ось, тепер ніяких лінощів. Список `coeffs` повинен бути виведеним на консоль повністю, а отже, обидва його елементи повинні бути обчислені до своєї нормальної форми, інакше ми не змогли б показати їх у консолі.

Ось філософія ледачої стратегії: навіть якщо нам потрібно обчислити вираз, ми обчислюємо його лише до тієї форми, яка є достатньою в конкретних умовах, і не більше.

Раціональність

Як вже було згадано, ледача стратегія допомагає програмі бути раціональною та не робити зайву роботу. Розглянемо приклад:

```
main :: IO ()
main = print $ take 5 evens
  where evens = [2, 4 .. 100]
```

Список `evens`, який формується через арифметичну послідовність, містить у собі парні числа від `2` до `100` включно. Використовується цей список в якості другого аргумента функції `take`, яка дає нам N перших елементів з переданого їй списку:

```
take   5         evens

візьми лише
       п'ять
       елементів з цього
                 списку
```

При запуску цієї програми ми отримаємо очікуваний результат:

```
[2,4,6,8,10]
```

У чому ж тут раціональність, запитаєте ви? А в тому, що список `evens` в результаті містив у собі лише 5 елементів. Так, але ж парних чисел від `2` до `100` набагато більше, ніж п'ять! Абсолютно вірно, але ледачість дозволяє нам зробити лише стільки роботи, скільки реально потрібно. Оскільки список `evens` потрібен лише функції `take`, яка, в свою чергу, хоче тільки п'ять перших його елементів — навіщо ж створювати елементи, які залишилися? Потрібно перші п'ять — отримай п'ять. Якщо ж напишемо так:

```
main :: IO ()
main = print $ take 50 evens
  where evens = [2, 4 .. 100]
```

тоді у списку `evens` виявиться вже п'ятдесят елементів, тому що саме стільки потрібно функції `take`. Повторю філософію ледачого раціоналізму: зробимо не стільки, скільки нам сказали, а лише стільки, скільки дійсно знадобиться.

Нескінченність

А що буде, якщо ми візьмемо зі списку `evens` 500 елементів? Ось так:

```
main :: IO ()
main = print $ take 500 evens
  where evens = [2, 4 .. 100]
```

Нічого страшного не трапиться, функція `take` перевіряє вихід за границі й у випадку, якщо її перший аргумент перевищує довжину списку, вона просто повертає нам той же список. Так, але ж ми хочемо побачити п'ятсот парних чисел, а не п'ятдесят! Можна було б й збільшити список:

```
main :: IO ()
main = print $ take 500 evens
  where evens = [2, 4 .. 100000]
```

але це ненадійно, адже потім знову може знадобитися ще більше. Потрібно щось універсальне, і в Haskell є відповідне рішення:

```haskell
main :: IO ()
main = print $ take 500 evens
  where evens = [2, 4 ..] -- Що це?
```

Тепер не сумнівайтеся: у списку `evens` буде не менше ніж п'ятисот парних чисел. Але що це за така конструкція? Початок дано, крок даний, а де ж кінець? Познайомтеся, це нескінченний список:

```haskell
[2, 4 ..]
```

Ледача модель обчислень дозволяє нам працювати з нескінченними структурами даних. Ось прямо так, починаючи з двійки, з кроком через один, йдемо в нескінченну далечінь… Жартую. Насправді, список вийде зовсім не нескінченним, а настільки великим, наскільки нам це знадобиться.

І справді, якщо функція `take` вимагає від нас N елементів — навіщо нам взагалі задавати закінчення діапазону списку? Все одно в ньому буде не більше ніж N. Нескінченна структура даних тим і корисна, що з неї завжди можна взяти стільки, скільки потрібно.

Звичайно, якщо б ми вирішили похуліганити:

```haskell
main :: IO ()
main = print evens -- Дай нам все!
  where evens = [2, 4 ..]
```

у цьому випадку в нашу консоль швидко посипалося б дуже багато чисел…

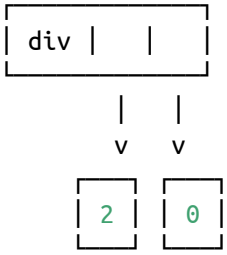
Space leak

Так, я повинен розповісти вам правду: є у ледачої стратегії обчислень темна сторона, що отримала назву space leak (букв. «витік простору»). І ось в чому її суть.

Згадаймо приклад з діленням:

```
main :: IO ()
main = print . strange $ 2 `div` 0
```

Як ми пам'ятаємо, ділення на нуль, так і не відбулося через непотрібність його результату. В цьому випадку вираз залишився у вигляді thunk. Виникає питання: що ж з ним сталось? У нас є функція `div` і є два значення типу `Int`, `2` та `0`. Якщо функція `div` так і не була застосована до них, де все це знаходилося в процесі роботи нашої програми? Воно знаходилося в пам'яті, у вигляді особливого графа, який можна зобразити так:



Тобто сама функція та два значення, які повинні були зайняти місце двох аргументів. І ось цей граф в пам'яті так і залишився незатребуваним. Здавалося б, в чому проблема? А проблема в кількості. Якщо ми змогли написати код, при роботі якого в пам'ять відклався один thunk, значить, теоретично, ми можемо написати і такий код, кількість thunk-ів при роботі якого буде обчислюватися мільйона-

ми. А враховуючи той факт, що кожен thunk займає в пам'яті хоча б декілька байт, ви можете собі уявити масштаб проблеми.

Причому ця проблема може виникнути з вельми безневинного на перший погляд коду:

```
bad :: [Int] -> Int -> Int
bad [] c = c
bad (_:others) c = bad others $ c + 1
```

Простенька рекурсивна функція, яка пробігає по непотрібному їй списку та збільшує свій другий аргумент на одиницю. Але я не просто так назвав її `bad`. Давайте застосуємо її:

```
bad [1, 2, 3] 0
```

Підставимо у визначення, яке містить зациклення:

```
bad (_: others) c = bad others $ c + 1

bad [1, 2, 3]   0 = bad [2, 3] $ 0 + 1

        ____              ______

                =                 =
```

«Голова» списку відкидається та ігнорується, а до `0` додається `1`. Але оскільки результат додавання поки що нікому не потрібен, додавання не виконується. Замість цього, на другій ітерації, ми бачимо наступне:

```
bad [2, 3] $ 0 + 1 = bad [3] $ (0 + 1) + 1
```

До попереднього виразу знову додається одиниця — й ми знову входимо в чергову ітерацію, так і не виконавши додавання:

```
bad [3] $ (0 + 1) + 1 = bad [] $ ((0 + 1) + 1) + 1
```

Отакої! Зупинилися на порожньому списку, згадуємо правило виходу з рекурсії:

```
bad [] c = c
```

Отже, у цьому випадку ми просто повертаємо значення другого аргументу. Зробимо це:

```
bad [] $ ((0 + 1) + 1) + 1 = ((0 + 1) + 1) + 1 = 3
```

І ось тільки тут ми реально обчислюємо другий аргумент, складаючи три одиниці. Ви запитаєте, чому ж ми накопичували ці додавання замість того, щоб робити їх відразу? Тому що ми ледачі: якщо результат додавання знадобився нам лише на останній ітерації, значить до цієї ітерації ніякого додавання не буде, адже ледачість змушує нас відкладати роботу до кінця.

Ось у цьому накопиченні й полягає вся біда. Уявімо, що ми написали так:

```
main :: IO ()
main = print $ bad [1..50000000] 0
```

50 мільйонів елементів, а значить, 50 мільйонів разів додавання другого аргументу з одиницею буде відкладатися, накопичуючи гігантський «хвіст» з (поки що) необчислених виразів. Хочете знати, що станеться при запуску такої програми? Її виконання, на MacBook Pro 2014 року, займе приблизно 63 секунди і з'їсть 6.4 ГБ пам'яті! А тепер уявіть, що сталося б, якби елементів у списку було не 50 мільйонів, а 50 мільярдів...

Іноді space leak помилково плутають з іншою проблемою, яка називається memory leak (англ. «витік пам'яті»), проте це зовсім не одне й те ж. Витік пам'яті — це помилка, характерна для мов з ручним керуванням пам'яттю, наприклад, C. Якщо ми виділимо пам'ять в купі (англ. heap), а потім втратимо вказівник, що зв'язує нас з цією пам'яттю — все, виділена пам'ять втекла, вона втрачена для нас навіки. Але у випадку space leak ми не втрачаємо пам'ять: коли весь цей «хвіст» з додавань зрештою обчислиться, пам'ять, зайнята мільйонами thunk-ів, звільниться. Ми не втрачаємо пам'ять, ми просто використовуємо її занадто багато.

Боротьба

Проблема space leak випливає з самої природи ледачих обчислень. Багато програмістів, дізнавшись про цю проблему, відвертаються від Haskell. Мовляв, якщо в цій мові можна легко написати код, який споживає купу пам'яті, значить ця мова точно не підходить для серйозного використання. Але не такий страшний чорт, як його малюють. Я розповім про два способи боротьби зі space leak.

Втім, з концептуальної точки зору спосіб всього один. Замислимося: якщо в наведеному вище прикладі ледачість стала причиною відкладання додавань на потім, що ж можна зробити? Відповідь проста: ми повинні прибрати зайву ледачість та замінити її строгістю. У цьому випадку застосування оператора додавання вже не буде відкладатися до останнього, а буде проводитися тут же, як в мовах зі строгою моделлю обчислень.

І як же ми можемо розбавити лінь строгістю? Ось два способи.

Оптимізація

Перший спосіб найпростіший — оптимізація. Коли компілятор перетворює наш код у програму, його можна попросити оптимізувати наш код, зробивши його більш ефективним, згідно тих чи інших критеріїв. Щоб попросити компілятор провести оптимізацію, ми повинні використовувати спеціальний параметр компілятора. Відкриємо збірочний файл нашого проекту `real.cabal`, знайдемо секцію `executable real-exe`, в якій є рядок:

```
ghc-options: ...
```

Цей рядок містить різні опції компілятора GHC, отож оптимізаційний параметр дописується саме сюди. Спробуємо підставити туди спочатку параметр `-O0`, а потім `-O2`. Результати запуску програми будуть такими:

```
Оптимізація Час    Пам'ять

-O0         63 с   6,4 ГБ

-O2         3.2 с  104 кБ
```

Вражаюча різниця, чи не так? Параметр `-O0` наказує компілятору не робити ніякої оптимізації, в цьому випадку кажуть про нульовий рівень оптимізації. Параметр `-O2`, навпаки, встановлює стандартний для production-проектів рівень оптимізації. Так от при стандартному рівні компілятор здатний розпізнати зайву ледачість в нашому коді і додати трохи жадібності. У прикладі вище компілятор побачить накопичення thunk-ів додавання та припинить його. Погодьтеся, з гігабайтів стрибнути відразу на кілобайти — це круто.

Так що, проблеми немає? Ну, якщо оптимізація `-O2` й так стандартна

— то давайте ставити її в наші проекти і забудемо про space leak! На жаль, не все так просто.

По-перше, компіляторна оптимізація це щось на кшталт чорної магії, на неї важко покладатися. Ми дуже вдячні компілятору GHC за спробу допомогти нам, але ця допомога не завжди відповідає нашим очікуванням. По-друге, на жаль, компілятор не завжди здатний розпізнати зайву ледачість в нашому коді, і в цьому випадку нам доводиться вдаватися до другого способу боротьби зі space leak.

Вручну

Повернемося до визначення функції `bad`:

```
bad :: [Int] -> Int -> Int
bad [] c = c
bad (_:others) c = bad others $ c + 1
```

Проблема, як ми вже зрозуміли, у другому аргументі:

```
bad others $ c + 1

             накопичення
             thunk-ів...
```

Перетворимо злу функцію в добру:

```
good :: [Int] -> Int -> Int
good [] c = c
good (_:others) c = good others $! c + 1
```

Цей код дасть нам приблизно такий же виграш, як оптимізація рівня `-O2`: секунди замість хвилин і кілобайти замість гігабайтів. Що ж

змінилося? Дивимося уважно:

```
good others $! c + 1

             ^
```

Замість звичного оператора застосування `$` ми бачимо оператор строгого застосування `$!` (англ. strict application operator). Цей оператор каже аргументу: «Забудь про лінощі, я наказую тобі негайно обчислитися до слабкої головної форми»:

```
good others $!      c + 1

                    обчисли цей
                    аргумент

            строго,
            а не
            ледаче!
```

Тому наш «хвіст» з thunk-ів не буде накопичуватися, адже на кожній з 50 мільйонів ітерацій буде відбуватися негайне застосування оператора додавання. Таким чином, змусити аргумент тут же обчислитися до слабкої головної або нормальної форми можна як за допомогою того, що цей аргумент прямо зараз комусь знадобився, так і за допомогою строгого застосування.

Ледачість і строгість разом

Функцію називають ледачою за тими аргументами, які не обчислюються, та строгою по тих аргументах, які обчислюються. Примітивний приклад:

```
fakeSum :: Int -> Int -> Int
fakeSum x _ = x + 100
```

Функція `fakeSum` строга за своїм першим аргументом і лінива за своїм другим аргументом. Перший аргумент `x` неодмінно буде обчислюватися, адже він передається оператору додавання. Другий аргумент ігнорується, залишившись необчисленим. До речі, існує простий спосіб перевірити, строга функція за деяким аргументом або ледача.

У стандартній бібліотеці Haskell визначена особлива функція `undefined`. Це — чорна діра: при спробі доторкнутися до неї програма гарантовано падає з помилкою. Перевіряємо:

```
main :: IO ()
main = print $ fakeSum 1 undefined
```

В цьому випадку ми отримаємо результат:

```
101
```

Чорна діра була проігнорована, адже функція `fakeSum` ледача по другому аргументу. Якщо ж ми напишемо так:

```
main :: IO ()
main = print $ fakeSum undefined 45
```

програма, спробувавши передати `undefined` оператору додавання, аварійно зупиниться. Або ось інший приклад:

```
main :: IO ()
main = print . head $ [23, undefined, undefined]
```

Не сумнівайтеся: програма спокійно поверне нам `23`, адже функція `head` строга лише по першому елементу переданого їй списку, інший його вміст її абсолютно не цікавить. Але якщо спробуєте витягнути другий або третій елемент з цього списку — крах неминучий.

Для допитливих

Haskell — не перша мову з ледачою стратегією обчислень. Відкрию вам історичний факт: у мови Haskell був попередник, мова програмування з красивим жіночим ім'ям Miranda. Ледачість і чиста функціональність прийшли в Haskell саме з Miranda, і лише у цих двох мовах ледача стратегія обчислення аргументів використовується за замовчуванням. На сьогоднішній день, наскільки мені відомо, мова Miranda мертва. Втім, як суто дослідницька мова вона, можливо, кимось і використовується.

Що ж стосується проблеми space leak, то на щастя, існують способи виявлення функцій, які є ненажерливими до пам'яті. Справді, уявіть собі великий проект, тисячі функцій, і щось їсть гігабайти пам'яті. Як знайти винного? Цей процес ще називають «space leak профілюванням». Розповідати про це тут я не стану, матеріал досить об'ємний. Але для особливо цікавих наводжу посилання на непогану англомовну статтю на тему: Chasing a Space Leak in Shake.

І ще пригадаймо ось це:

```
square     x   =        x    *    x
         /   \        /   \     /   \
square $ 2 + 2 =     (2 + 2) * (2 + 2) = 16

         обчислюємо  і що,
                     знову
                     обчислюємо?!
```

Уважний читач здивується, мовляв, невже вираз `2 + 2` обчислюється двічі?! Адже це нераціонально. Звичайно нераціонально, тому в реальності воно буде обчислено один раз. В Haskell є особливий механізм «шарінгу» (англ. sharing), що дозволяє уникнути марної роботи. Якщо у нас є декілька однакових виразів, то їх обчислення відбувається один раз, а результат зберігається та потім просто підставляється в потрібні місця. Наприклад:

```
main :: IO ()
main =
  let x = sin 2 in print x * x
```

Якби не sharing-механізм, функція `sin` була б застосована до `2` двічі. На щастя, значення синуса буде обчислено раз і тут же збережено, щоб потім просто стати на місця тих двох `x`.

Розділ 19

Наші типи

Ось ми й дісталися до Другого Кита Haskell — до **Типів**. Звісно, ми працювали з типами майже з самого початку, але вам вже добряче набридли всі ці `Int` та `String`, чи не так? Прийшла пора познайомитися з типами значно ближче.

Знайомство

Дивно, але в Haskell дуже мало вбудованих типів, тобто таких, про які компілятор знає з самого початку. Є `Int`, є `Double`, `Char`, ну і ще кілька. Всі ж інші типи, які навіть носять статус стандартних, не є вбудованими в мову. Замість цього вони визначені в стандартній чи інших бібліотеках, причому визначені точно так само, як ми будемо визначати і наші власні типи. А оскільки без своїх типів написати якусь серйозну програму в нас не вийде, ця тема гідна найпильнішого погляду.

Визначимо тип `Transport` для двох відомих протоколів транспортного рівня моделі OSI:

```
data Transport = TCP | UDP
```

Перед нами — дуже простий, але вже наш власний тип. Розглянемо його уважніше.

Ключове слово `data` — це початок визначення типу. Далі йде назва типу, в даному випадку `Transport`. Ім'я будь-якого типу повинно починатися з великої літери. Потім йде знак рівності, після якого починається фактичний опис типу, його «тіло». В даному випадку воно складається з двох найпростіших конструкторів. Конструктор значення (англ. data constructor) — це те, що конструює значення даного типу. Тут у нас два конструктора, `TCP` і `UDP`, кожен з яких конструює значення типу `Transport`. Ім'я конструктора теж повинно починатися з великої літери. Іноді для лаконічності конструктор значення називають просто конструктором.

Таке визначення легко читається:

```
data Transport =  TCP |   UDP

тип  Transport це TCP або UDP
```

Тепер ми можемо використовувати тип `Transport`, тобто створювати значення цього типу і щось з ними робити. Наприклад, в `let`-виразі:

```
let protocol = TCP
```

Ми створили значення `protocol` типу `Transport`, використавши конструктор `TCP`. А можна й так:

```
let protocol = UDP
```

Хоча ми використовували різні конструктори, тип значення `protocol` в обох випадках один і той самий — `Transport`.

Розширити подібний тип дуже просто. Додамо новий протокол SCTP (Stream Transmission Control Protocol):

```
data Transport = TCP | UDP | SCTP
```

Третій конструктор значення дав нам третій спосіб створення значення типу `Transport`.

Значення-пустушка

Замислимося: коли ми говоримо про значення типу `Transport` — про що насправді йде мова? Здавалося б, значення фактично немає: ні числа, ні рядка, просто три конструктори. Так ось вони і є значеннями. Коли ми пишемо:

```
let protocol = SCTP
```

ми створюємо значення типу `Transport` з конкретним вмістом у вигляді `SCTP`. Конструктор — це і є вміст. Даний вид конструктора називається нульарним (англ. nullary). Тип `Transport` має три нульарних конструктори. І навіть такий простий тип вже може бути нам корисним:

```
checkProtocol :: Transport -> String
checkProtocol transport = case of transport
  TCP -> ”that’s TCP protocol.”
  UDP -> ”that’s UDP protocol.”
  SCTP -> ”that’s SCTP protocol.”

main :: IO ()
main = putStrLn . checkProtocol $ TCP
```

В результаті побачимо:

```
That’s protocol TCP.
```

Функція `checkProtocol` оголошена як приймаюча аргумент типу `Transport`, а застосовується вона до значення, породженому конструктором `TCP`. В даному випадку конструкція `case-of` порівнює аргумент з конструкторами. Саме тому нам не потрібна функція `otherwise`, адже ніяким іншим способом, окрім як за допомогою трьох конструкторів, значення типу `Transport` створити неможливо, а значить, один з конструкторів гарантовано співпаде з аргументом.

Тип, що складається лише з нульарных конструкторів, називають ще переліком (англ. enumeration). Конструкторів може бути скільки завгодно, в тому числі один-єдиний (хоча користь від такого типу була б невелика). Ось ще один відомий приклад:

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

Зверніть увагу на форматування, коли ментальні «АБО» вирівняні строго під знаком рівності. Такий стиль ви зустрінете в багатьох реальних Haskell-проектах.

Значення типу `Day` відображено одним з семи конструкторів. Давайте щось з ними зробимо:

```haskell
data WorkMode = FiveDays | SixDays

workingDays :: WorkMode -> [Day]
workingDays FiveDays = [ Monday
                       , Tuesday
                       , Wednesday
                       , Thursday
                       , Friday
                       ]
workingDays SixDays = [ Monday
                      , Tuesday
                      , Wednesday
                      , Thursday
                      , Friday
                      , Saturday
                      ]
```

Функція `workingDays` повертає список типу `[Day]`, і в разі п'ятиденного робочого тижня, відображеного конструктором `FiveDays`, цей пере-

лік сформований п’ятьма конструкторами, а в разі шестиденного — шістьма конструкторами.

Користь від типів, сформованих нульарными конструкторами, не дуже велика, хоча зустрічатися з такими типами ви будете часто.

Відкрию секрет: новий тип можна визначити не тільки за допомогою ключового слова `data`, але про це дізнаємося в одному з наступних розділів.

А тепер ми можемо познайомитися з більш корисними типами.

Розділ 20

Алгебраїчні Типи Даних

АТД, або Алгебраїчні Типи Даних (англ. ADT, Algebraic Data Type), займають почесне місце в світі типів Haskell. Абсолютно переважна більшість ваших власних типів будуть алгебраїчними, і те саме можна сказати про типи з безлічі Haskell-пакетів. Алгебраїчним типом даних називають такий тип, який складений з інших типів. Ми беремо прості типи й будуємо з них, як із цеглин, типи складні, а з них — ще більш складні. Це дає нам неймовірний простір для творчості.

Залишимо мережеві протоколи і дні тижня, розглянемо такий приклад:

```
data IPAddress = IPAddress String
```

Тип `IPAddress` використовує один-єдиний конструктор значення, але дещо змінилося. По-перше, імена типу і конструктора збігаються. Це цілком легально, ви зустрінете таке не раз. По-друге, конструктор вже не нульарный, а унарний (англ. unary), тому що тепер він пов'язаний з одним значенням типу `String`. І ось як створюються значення типу `IPAddress`:

```
let ip = IPAddress "127.0.0.1"
```

Значення `ip` типу `IPAddress` утворено конструктором і конкретним значенням якогось типу:

```
let ip = IPAddress       "127.0.0.1"

         конструктор     значення
         значення типу   типу
         IPAddress       String

         | значення типу IPAddress |
```

Значення всередині нашого типу називають ще полем (англ. field):

```
data IPAddress = IPAddress   String

     тип         конструктор поле
```

Розширимо тип `IPAddress`, зробивши його більш сучасним:

```
data IPAddress = IPv4 String | IPv6 String
```

Тепер у нас два конструктори, які відповідають різним IP-версіям. Це дозволить нам створювати значення типу `IPAddress` так:

```
let ip = IPv4 "127.0.0.1"
```

або так:

```
let ip = IPv6 "2001:0 db8:0000:0042:0000:8a2e:0370:7334"
```

Зробимо тип ще більш зручним. Так, при роботі з IP-адресою нам часто потрібен `localhost`. І щоб явно не писати "`127.0.0.1`" і "`0:0:0:0:0:0:0:1`", введемо ще два конструктори:

```
data IPAddress = IPv4 String
               | IPv4Localhost
               | IPv6 String
               | IPv6Localhost
```

Оскільки значення `localhost` нам заздалегідь відомі, немає потреби вказувати явно. Замість цього, коли нам знадобиться `IPv4-localhost`, пишемо так:

```
let ip = IPv4Localhost
```

Витягуємо значення

Припустимо, ми створили значення `google`:

```
let google = IPv4 "173.194.122.194"
```

Як же нам потім отримати конкретне значення рядка з `google`? За допомогою нашого старого друга, патерн матчінгу:

```
checkIP :: IPAddress -> String
checkIP (IPv4 address) = "IP is '" ++ address ++ "'."

main :: IO ()
main = putStrLn . checkIP $ IPv4 "173.194.122.194"
```

Результат:

```
IP is '173.194.122.194'.
```

Поглянемо на визначення:

```
checkIP (IPv4 address) = "IP is '" ++ address ++ "'."
```

Тут ми говоримо: «Ми знаємо, що значення типу `IPAddress` сформовано з конструктором і рядком». Однак уважний компілятор зробить нам зауваження:

```
Match Pattern(es) are non-exhaustive
In an equation for 'checkIP':
Patterns not matched:
IPv4Localhost
IPv6 _
IPv6Localhost
```

Справді, звідки ми знаємо, що значення, до якого застосували функцію `checkIP`, було сформовано саме з допомогою конструктора `IPv4`? У нас же є ще три конструктори, і нам слід перевірити їх всіх:

```
checkIP :: IPAddress -> String
checkIP (IPv4 address) = "IPv4 is '" ++ address ++ "'."
checkIP IPv4Localhost = "IPv4, localhost."
checkIP (IPv6 address) = "IPv6 is '" ++ address ++ "'."
checkIP IPv6Localhost = "IPv6, localhost."
```

З яким конструктором співпало — з таким і було створено значення. Можна, звичайно, і так перевірити:

```
checkIP :: IPAddress -> String
checkIP addr = case of addr
  IPv4 address -> ”IPv4 is ’” ++ address ++ ”’.”
  IPv4Localhost -> ”IPv4, localhost.”
  IPv6 address -> ”IPv6 is ’” ++ address ++ ”’.”
  IPv6Localhost -> ”IPv6, localhost.”
```

Конструюємо

Визначимо тип для мережевої точки:

```
data Security = EndPoint String Int
```

Конструктор `Security` — бінарний, адже тут вже два значення. Створюємо звичайним способом:

```
let googlePoint = EndPoint ”173.194.122.194” 80
```

Конкретні значення витягаємо знову-таки через патерн матчінг:

```
main :: IO ()
main = putStrLn $ ”The host is:” ++ host
  where
    EndPoint host _ = EndPoint ”173.194.122.194” 80

    |-- зразок  --|   |-------- значення ---------|
```

Зверніть увагу, що друге поле, відповідне порту, відображено універсальним зразком _, тому що в даному випадку нас цікавить тільки значення хоста, а порт просто ігнорується.

І все було б добре, але тип `Security` мені не дуже подобається. Є в ньому щось негарне. Першим полем виступає рядок, що містить IP-адресу, але навіщо нам рядок? У нас є прекрасний тип `IPAddress`, він кращий ніж безликий рядок. Це загальне правило для Haskell-розробника: чим більше інформації несе в собі тип, тим він кращий. Давайте змінимо визначення:

```
data Security = EndPoint IPAddress Int
```

Тип став зрозумілішим, і ось як ми тепер будемо створювати значення:

```
let google = EndPoint (IPv4 ”173.194.122.194”) 80
```

Красиво. Отримувати конкретні значення будемо так:

```
main :: IO ()
main = putStrLn $ ”The host is:” ++ ip
  where
    EndPoint (IPv4 ip) _ = EndPoint (IPv4 ”173.194.122.194”) 80
                ____                     ____

                    ==                       =================
```

Тут ми знову-таки ігноруємо порт, але значення IP-адреси витягаємо вже на основі зразка з конструктором `IPv4`.

Це простий приклад того, як з простих типів утворюються більш складні. Але складний тип зовсім не означає складну роботу з ним, патерн матчінг елегантний як завжди. А незабаром ми дізнаємося про інший спосіб роботи з полями типів, без патерн матчінгу.

Цікаво, що конструктори типів теж можна компонувати, погляньте:

```
main :: IO ()
main = putStrLn $ "The host is:" ++ ip
  where
    EndPoint (IPv4 ip) _ = (EndPoint . IPv4 $ "173.194.122.194") 80
```

Це схоже на маленьке диво, але конструктори типів можна компонувати за допомогою знайомого нам оператора композиції функцій:

```
(EndPoint . IPv4 $ "173.194.122.194") 80

            |     значення типу     |
            |_____ IPAddress _____|
```

Вам це нічого не нагадує? Це ж так само, як ми працювали з функціями! З цього ми робимо висновок: конструктор значення можна розглядати як особливу функцію. Насправді:

```
EndPoint  (IPv4 "173.194.122.194")  80

"функція" |        перший         |  другий
          |______ аргумент ______|  аргумент
```

Ми ніби застосовуємо конструктор до конкретних значень як до аргументів, в результаті чого отримуємо значення нашого типу. А раз так, ми можемо компонувати конструктори так само, як і звичайні функції, лише б їх типи були комбінованими. В даному випадку все добре: тип значення, що повертається конструктором `IPv4`, збігається з типом першого аргумента конструктора `Security`.

Ось ми й познайомилися з цими типами. Настав час дізнатися про більш зручну роботу з полями типів.

Розділ 21

АТД: поля з мітками

Багато типів в реальних проектах досить великі. Погляньте:

```
data Arguments = Arguments Port
                           Endpoint
                           RedirectData
                           FilePath
                           FilePath
                           Bool
                           FilePath
```

Значення типу `Arguments` зберігає в своїх полях деякі значення, отримані з параметрів командного рядка, з якими запущена одна з моїх програм. І все було б добре, але працювати з таким типом абсолютно незручно. Він містить сім полів, і патерн матчінг був би занадто громіздким, уявіть собі:

```
...
where
Arguments _ _ _ redirectLib _ _ xpi = arguments
```

Більше того, коли ми дивимося на визначення типу, призначення полів залишається таємницею за сімома печатями. Бачите передостаннє поле? Воно має тип `Bool` і, ясна річ, відображає якийсь прапорець. Але що це за прапорець, читач не уявляє. На щастя, існує спосіб, який рятує нас від обох цих проблем.

Мітки

Ми можемо забезпечити наші поля мітками (англ. label). Ось як це виглядає:

```
data Arguments = Arguments { runWDServer :: Port
                           , withWDServer :: Endpoint
                           , redirect :: RedirectData
                           , redirectLib :: FilePath
                           , screenshotsDir :: FilePath
                           , noScreenshots :: Bool
                           , harWithXPI :: FilePath
                           }
```

Тепер призначення міток більш зрозуміле. Схема визначення така:

```
data Arguments = Arguments   { runWDServer :: Port }

тип  такий-то    конструктор   мітка поля     тип
                                              поля
```

Тепер поле має не тільки тип, але й назву, що робить наше визначення значно більш читабельним. Поля в цьому випадку розділені комами та знаходяться в фігурних дужках.

Якщо два чи більше полів одного типу йдуть поспіль, тип можна

вказати лише для останньої з міток. А отже, якщо у нас є ось такий тип:

```
data Patient = Patient { firstName :: String
                       , lastName :: String
                       , email :: String
                       }
```

його визначення можна трохи спростити і написати так:

```
data Patient = Patient { firstName
                       , lastName
                       , email :: String
                       }
```

Оскільки тип всіх трьох полів однаковий, ми вказуємо його лише для останньої з міток. Ще приклад повної форми:

```
data Patient = Patient { firstName :: String
                       , lastName :: String
                       , email :: String
                       , age :: Int
                       , diseaseId :: Int
                       , isIndoor :: Bool
                       , hasInsurance :: Bool
                       }
```

і відразу спрощуємо:

```
data Patient = Patient { firstName
                       , lastName
                       , email :: String
                       , age
                       , diseaseId :: Int
                       , isIndoor
                       , hasInsurance :: Bool
                       }
```

Поля `firstName`, `lastName` та `email` мають тип `String`, поля `age` і `diseaseId` — тип `Int`, і два поля — тип `Bool`.

Getter і Setter?

Що ж таке мітки? Фактично, це особливі функції, згенеровані автоматично. Ці функції мають три призначення: створювати, вилучати й змінювати. Так, я не обмовився, змінювати. Але про це трохи пізніше, нехай буде маленька інтрига.

Ось як ми створюємо значення типу `Patient`

```
main :: IO ()
main = print $ diseaseId patient
  where
  patient = Patient {
    firstName = "John"
    , lastName = "Doe"
    , email = "john.doe@gmail.com"
    , age = 24
    , diseaseId = 431
    , isIndoor = True
    , hasInsurance = True
    }
```

Мітки полів використовуються як свого роду setter (від англ. set, «встановлювати»):

```
patient = Patient { firstName    =      "John"
у цьому   типу      поле з
значення  Patient   цією міткою  рівне  цьому рядку
```

Крім того, мітку можна використовувати і як getter (від англ. get, «отримувати»):

```
main = print $ diseaseId   patient

               мітка як    аргумент
               функція
```

Ми застосовуємо мітку до значення типу `Patient` і отримуємо значення відповідного цій мітці поля. Тому для отримання значень полів нам уже не потрібен патерн матчінг.

Але що ж за інтригу я приготував насамкінець? Попередньо я згадав, що мітки використовуються не тільки для задання значень по-

лів і для їх отримання, але і для зміни. Ось що я мав на увазі:

```
main :: IO ()
main = print $ email patientWithChangedEmail
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com" -- Змінюємо???
    }

    patient = Patient {
      firstName = "John"
      , lastName = "Doe"
      , email = "john.doe@gmail.com"
      , age = 24
      , diseaseId = 431
      , isIndoor = True
      , hasInsurance = True
    }
```

При запуску програми отримаємо:

```
j.d@gmail.com
```

Але постривайте, що ж тут сталося? Адже в Haskell, як ми знаємо, немає оператора присвоювання, однак значення поля з міткою `email` змінилося. Пам'ятаю, коли я вперше побачив подібний приклад, то дуже здивувався, мовляв, чи не ввели мене в оману щодо незмінності значень в Haskell?!

Ні, не ввели. Подібний запис:

```
patientWithChangedEmail = patient {
  email = "j.d@gmail.com"
}
```

дійсно схожий на зміну поля через присвоєння йому нового значе-

ння, але в дійсності ніякої зміни не відбулося. Коли я назвав мітку setter-ом, я трохи злукавив, адже класичний setter зі світу ООП був би неможливий в Haskell. Подивимося ще раз уважніше:

```
...
where
  patientWithChangedEmail = patient {
    email = "j.d@gmail.com" -- Змінюємо???
  }

  patient = Patient {
    firstName = "John"
    , lastName = "Doe"
    , email = "john.doe@gmail.com"
    , age = 24
    , diseaseId = 431
    , isIndoor = True
    , hasInsurance = True
  }
```

Погляньте, адже у нас тепер два значення типу `Patient`, `patient` і `patientWithChangedEmail`. Ці значення не мають одне до одного жодного відношення. Згадайте, як я говорив, що в Haskell не можна змінити наявні значення, а можна лише створити на основі наявного нове значення. Це саме те, що тут сталося: ми взяли наявне значення `patient` і на його основі створили вже нове значення `patientWithChangedEmail`, значення поля `email` в якому тепер інше. Зрозуміло, що поле `email` у значенні `patient` залишилося незмінним.

Будьте уважні при ініціалізації значення з полями: ви зобов'язані надати значення для всіх полів. Якщо ви напишете так:

```
main :: IO ()
main = print $ email patientWithChangedEmail
  where
    patientWithChangedEmail = patient {
      email = "j.d@gmail.com" -- Змінюємо???
    }

    patient = Patient {
      firstName = "John"
      , lastName = "Doe"
      , email = "john.doe@gmail.com"
      , age = 24
      , diseaseId = 431
      , isIndoor = True
    }

    -- Поле hasInsurance забули!
```

код скомпілюється, але уважний компілятор попередить вас про проблему:

```
Fields of 'Patient' not initialised: hasInsurance
```

Будь ласка, не нехтуйте подібним попередженням, адже якщо ви проігноруєте його і потім спробуєте звернутися до неініціалізованого поля:

```
main = print $ hasInsurance patient
...
```

ваша програма аварійно завершиться на етапі виконання з очікуваною помилкою:

```
Missing record field in construction hasInsurance
```

Не забувайте: компілятор — ваш добрий друг.

Без міток

Пам'ятайте, що мітки полів — це синтаксичний цукор (англ. syntactic sugar), і ми можемо обійтися без нього. Навіть якщо тип був визначений з мітками, як наш `Patient`, ми можемо працювати з ним по-старому:

```
data Patient = Patient { firstName :: String
                       , lastName :: String
                       , email :: String
                       , age :: Int
                       , diseaseId :: Int
                       , isIndoor :: Bool
                       , hasInsurance :: Bool
                       }

main :: IO ()
main = print $ hasInsurance patient
  where
    -- Створюємо по-старому...
    patient = Patient "John"
                      "Doe"
                      "john.doe@gmail.com"
                      24
                      431
                      True
                      True
```

Відповідно, отримувати значення полів теж можна по-старому, через патерн матчінг:

```
main :: IO ()
main = print insurance
  where
    -- Страшенно незручно, але якщо бажаєте...
    Patient _ _ _ _ _ _ insurance = patient
    patient = Patient "John"
                      "Doe"
                      "john.doe@gmail.com"
                      24
                      431
                      True
                      True
```

З поняттям «синтаксичний цукор» ми зустрінемося ще не раз, на більш складних прикладах.

Розділ 22

Новий тип

Крім `data` існує ще одне ключове слово, призначене для визначення нового типу. Воно так і називається — `newtype`. Ці слова схожі одне на одне «в односторонньому порядку »: ви можете поставити `data` на місце `newtype`, але не навпаки.

Відмінності

Тип, який визначається за допомогою слова `newtype`, зобов'язаний мати один і тільки один конструктор значення. Ми можемо написати так:

```
newtype IPAddress = IP String
```

А ось так не можемо:

```
newtype IPAddress = IP String | Localhost
```

Компілятор буде проти:

```
A newtype must have exactly one constructor,
  but 'IPAddress' has two
In the newtype declaration for 'IPAddress'
```

Окрім того, в такому типі має бути одне і тільки одне поле. Тобто можна так:

```
newtype IPAddress = IP String
```

Або так, з міткою:

```
newtype IPAddress = IP { value :: String }
```

А ось два або більше полів запхати не вдасться:

```
newtype EndPoint = EndPoint String Int
```

Компілятор знову зверне нашу увагу на проблему:

```
The constructor of a newtype must have exactly one field
  but 'Security' has two
In the definition of data constructor 'Security'
In the newtype declaration for 'Security'
```

Ба навіть більше, нульарный конструктор теж не підійде:

```
newtype HardDay = Monday
```

І знову помилка:

```
The constructor of a newtype must have exactly one field
  but 'Monday' has none
```

Навіщо він потрібен?

Справді, навіщо нам потрібна така річ? Це не можна, те не можна. В чому сенс?

Сенс в оптимізації. Зверніть увагу на модель `newtype`:

```
newtype IPAddress = IP          String

новий   назва       конструктор поле
тип                 значення
```

Фактично, `newtype` бере одне значення певного існуючого типу та лише загортає його у свій конструктор. Саме тому тип, введений за допомогою `newtype`, не відноситься до АТД, та з точки зору компілятора він є лише перейменуванням типу (англ. type renaming). Це робить такий тип більш простим та ефективним з точки зору представлення в пам'яті, аніж тип, який визначається за допомогою `data`.

Коли ми пишемо так:

```
data IPAddress = IP String
```

ми кажемо компілятору: «`IPAddress` — це абсолютно новий та самобутній тип, якого ніколи не було раніше». А коли пишемо так:

```
newtype IPAddress = IP String
```

ми кажемо: «`IPAddress` — це лише обгортка для значення вже існуючого типу `String`».

Ось ми і познайомилися з `newtype`-типами. У багатьох пакетах ви не раз зустрінете такі типи.

Для допитливих

Уважний читач запитає, в чому ж фундаментальна відмінність типів, які створюються за допомогою `newtype`, від типів, що вводяться з допомогою `type`? Там синонім, тут — обгортка. Відмінність ось у чому.

Коли ми пишемо так:

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

ми оголошуємо: «Тип `String` — це еквівалентна заміна типу `[Char]`». Тому скрізь, де в коді стоїть `[Char]`, ми можемо поставити `String`, а всюди, де стоїть `String`, ми можемо поставити `[Char]`. Наприклад, якщо функція оголошена так:

```
replace :: String
        -> String
        -> String
        -> String
```

ми можемо спокійно переписати оголошення:

```
replace :: [Char]
        -> [Char]
        -> [Char]
        -> [Char]
```

і нічого не зміниться.

Якщо ж ми пишемо так:

```
newtype MyInt = MyInt Int
```

ми оголошуємо: «Тип `MyInt` — це новий тип, представлення якого таке ж, як у типу `Int`». Ми не можемо просто взяти й поставити `MyInt` на місце `Int`, тому що ці типи рівні лише з точки зору представлення в пам'яті, а з точки зору системи типів вони абсолютно різні. # Далі буде…

Робота над книгою йде повним ходом, на вас чекає ще багато цікавого! Слідкуйте за новинами про оновлення в нашому чаті та в моєму Твіттері.