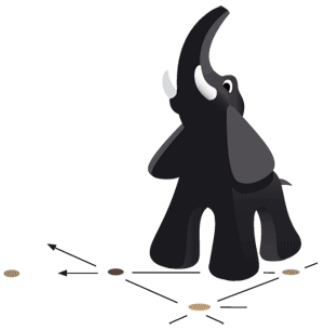
Вивчить собі Хаскела на велике щастя!

Автор: Міран Ліповача

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

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

Переклад виконано за підтримки
Словненя



2017-05-21T00:06:19Z
Версія v4.7-54-gda41cf2

Зміст

Розділ 4

Синтаксис у функціях

Переклад українською Ганни Лелів

4.1 Зіставлення із взірцем

Цей розділ розповість про кілька класних синтаксичних структур Хаскела, і ми розпочнемо із зіставлення із взірцем. Зіставлення із взірцем складається із означування взірців, яким повинні відповідати дані, перевірки чи дані дійсно відповідають цим взірцям, і, врешті, деконструювання даних відповідно до тих взірців.

Означуючи функції, можна означити різні тіла функцій для різних взірців. Це зазвичай веде до гарного — простого і читабельного — коду. Із взірцем можна зіставляти будь-які дані — числа, символи, списки, кортежі, тощо. Напишімо тривіальну функцію, яка перевіряє, чи число, яке ми надали, сімірка чи ні.

```
lucky :: Integral a => a -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

Коли ви викличете `lucky`, взірці перевірятимуться зверху до низу, і коли зіставлення буде успішним, буде використано відповідне тіло функції. Єдиний спосіб, в який число може зіставитись із першим взірцем, це бути 7. Якщо ж число не є 7, пошук перейде до другого взірця, який підходить до будь-чого і прив'яже те будь-що до імені `x`. Цю фун-

кцію також можна було б реалізувати, використавши інструкцію розгалуження. А що, якби ми захотіли мати функцію, яка впізнає числа від 1 до 5 і видає `"Not between 1 and 5"` для решти чисел? Якби не було ідіоми зіставлення із взірцем довелося б написати доволі заплутане дерево вибору «якщо-тоді-інакше». Однак, із нею, маємо:

```
sayMe :: Integral a => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

Зауважте, що якби ми поставили останній взірець (який усе ловить) першим, то завжди отримували б напис `"Not between 1 and 5"`, адже він ловив би всі числа, і в них не було б можливості «провалитися» до наступних взірців і пройти зіставлення з ними.

Пригадуєте функцію факторіалу, яку ми реалізували раніше? Ми означили факторіал числа `n` як `product [1..n]`. Ми також можемо означити функцію факторіалу *рекурсивно* — так, як її зазвичай означують у математиці. Спершу ми вказуємо, що факторіал 0 є 1. Тоді ми означуємо, що факторіал будь-якого додатного цілого числа є це число помножене на факторіал його попередника. Ось як це виглядає мовою Хаскела:

```
factorial :: Integral a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Ми вперше означили функцію рекурсивно. У Хаскелі рекурсія дуже важлива, тому згодом ми розглянемо її детальніше. Але, у двох словах, ось що трапиться, коли ми спробуємо добути факторіал числа 3, наприклад. Розрахунки почнуться з `3 * factorial 2`. Факторіал 2 — `2 * factorial 1`, отож наразі маємо `3 * (2 * factorial 1)`. `factorial 1` дорівнює `1 * factorial 0`, отож ми отримуємо `3 * (2 * (1 * factorial 0))`. І ось ми дійшли до спритного трюку — ми означили факторіал 0 як 1, і оскільки ми натрапляємо на цей взірець першим, до універсального взірця справа не доходить, і тому повертається 1. Отож, остаточний результат є еквівалентним `3 * (2 * (1 * 1))`. Якби ми написали другий взірець вгорі над першим, він би ловив усі числа, 0 включно, і наші обчислення ніколи б не завершилися. Ось чому коли ми означуємо взірці, так важливо подавати означення у правильному порядку і означувати найбільш конкретні взірці першими, а більш загальні — пізніше.

Зіставлення із взірцем може закінчитися невдачею. Якщо ми означимо функцію ось так:

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

а тоді спробуємо викликати її із вхідними даними, про які ми не подбали, ось що трапиться:

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
*** Exception: tut.hs:(53,0)-(55,21):
    Non-exhaustive patterns in function charName
```

Справедливі нарікання на невичерпуючі взірці. Створюючи групу взірців, ми завжди мусимо включити до неї універсальний взірець, щоб наша програма не завершилася аварійно через неочікувані вхідні дані.

Зіставлення із взірцем можна використати і для кортежів. А що якби ми захотіли створити функцію, яка отримує два вектори у 2D просторі (і вектор представлено у формі пари) і додає їх? Щоб додати два вектори, ми окремо додаємо їхні компоненти x, а тоді окремо їхні компоненти y. Ось як би ми зробили, якби ми нічого не знали про зіставлення із взірцями:

```
addVectors :: Num a => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Так, працює. Але існує кращий спосіб. Спробуймо змінити цю функцію так, щоб вона використовувала зіставлення із взірцем.

```
addVectors :: Num a => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Ось так! У сто разів краще. Зверніть увагу що це вже і є універсальний взірець. Тип `addVectors` (в обох випадках) — це `addVectors :: Num a => (a, a) -> (a, a) -> (a, a)`, отож ми точно отримаємо дві пари як параметри.

`fst` and `snd` виокремлюють компоненти пар. А як щодо трійок? Наразі ми не маємо жодних готових функцій для трійок, але можемо написати свої власні.

```
first :: (a, b, c) -> a
```

```
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z
```

`_` означає те ж саме, що й у спискових характерах: нам байдуже що воно таке та частина, і тому ми просто пишемо `_`.

А це нагадує мені, що зіставляти із взірцями можна і в спискових характерах. Ось, погляньте:

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a + b | (a,b) <- xs]
[4,7,6,8,11,4]
```

Якщо зіставлення зі взірцем зазнає невдачі, то воно просто перейде до наступного елемента.

Списки також можна використати у зіставленні із взірцем. Можна зіставити з порожнім списком `[]` чи будь-яким взірцем, куди входить `:` і порожній список. Але оскільки `[1,2,3]` — це просто синтаксичний цукор для `1:2:3:[]`, ви можете використати вже відомий нам взірець. Такий взірець, як-от `x:xs`, прив'яже початок списку до `x`, а решту — до `xs`. Якщо у списку буде тільки один елемент, то `xs` буде зв'язано із порожнім списком.

Примітка: Взірець `x:xs` часто використовують, особливо з рекурсивними функціями. Але взірці, які містять `:`, можна зіставити тільки зі списками, що мають довжину 1 чи більше.

Якщо ви, скажімо, хочете прив'язати перші три елементи до змінних, а решту списку до іншої змінної, ви можете скористатися `x:y:z:zs`. Його можна зіставити тільки зі списками, які містять три і більше елементів.

Тепер ми вже знаємо, як зіставляти списки зі взірцями, тому спробуймо реалізувати нашу власну функцію `head`.

```
head' :: [a] -> a
head' []    = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

Перевіримо, чи працює:

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

Супер! Зауважте, що, якщо ви хочете прив'язати до кілька змінних (навіть якщо одна з них — це просто підкреслення `_`, яке ні до чого не прив'язується), ми повинні взяти все в дужки. Зверніть також увагу на функцію `error`, яку ми використали. Вона бере рядок і генерує помилку виконання, використовуючи той рядок як інформацію про те, яка саме помилка трапилася. Через неї програма завершується аварійно, тому не радимо використовувати цю функцію занадто часто. Але викликати `head` із порожнім списком — безглуздо.

Напишімо тривіальну функцію, яка розповідає нам про кілька перших елементів списку у (не)зручній формі англійською.

```
tell :: Show a => [a] -> String
tell []       = "The list is empty"
tell (x:[])   = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and "
                ++ show y
tell (x:y:_)  = "This list is long. The first two elements are: "
                ++ show x ++ " and " ++ show y
```

Ця функція безпечна, оскільки вона працює з порожнім списком, одноелементним списком, списком із двома елементами і списком із більш ніж двома елементами. Зверніть увагу, що `(x:[])` і `(x:y:[])` можна переписати як `[x]` і `[x,y]`, відповідно (оскільки це синтаксичний цукор, тут нам дужки не потрібні). Ми не можемо переписати `(x:y:_)` із квадратними дужками, оскільки його можна зіставити з будь-яким списком, завдовжки 2 чи більше.

Ми вже реалізували нашу власну функцію `length` за допомогою спискових характерів. Зараз ми спробуємо зробити те саме, використовуючи зіставлення із взірцем і невеличку рекурсію:

```
length' :: Num b => [a] -> b
length' []     = 0
length' (_:xs) = 1 + length' xs
```

Це нагадує функцію факторіалу, яку ми написали раніше. Спочатку ми означили результат для відомих **вхідних даних** [input data] — порожнього списку. Цей взірець також відомий під назвою граничної умови. Тоді в другому взірці ми розбираємо список, поділивши його на голову і хвіст. Ми вказуємо, що довжина дорівнює 1 плюс довжина хвоста. `_` зіставляється з головою, бо нам, по

суті, байдуже, що то є. Зверніть увагу що ми подбали про всі можливі варіанти для списку: перший взірець зіставляється із порожнім списком, а другий — з усім, що не є порожнім списком.

Погляньмо, що трапиться, якщо ми викличемо `length'` по `"ham"`. Спершу воно перевірить, чи це не порожній список. Оскільки список не порожній, перехід буде здійснено до другого взірця. Далі відбувається зіставлення із другим взірцем, а він стверджує, що довжина дорівнює `1 + length' "am"`, оскільки ми поділили список на голову і хвіст і відкинули голову. Гаразд. `length'` від `"am"` дорівнює `1 + length' "m"`. Отож, ми маємо `1 + (1 + length' "m")`. `length' "m"` дорівнює `1 + length' ""` (можемо також написати `1 + length' []`). І ми означили `length' []` як `0`. Тому врешті-решт ми отримуємо `1 + (1 + (1 + 0))`.

Реалізуймо `sum`. Ми знаємо, що сума порожнього списку дорівнює 0. Ми записуємо це першим взірцем. Нам також відомо, що сума списку — це голова плюс сума решти списку. Якщо ми це все запишемо разом, отримаємо:

```
sum' :: Num a => [a] -> a
sum' []     = 0
sum' (x:xs) = x + sum' xs
```

Є ще така штука як *взірці із ім'ям*. Це зручний спосіб розвалити щось на складові відповідно до взірця і поіменувати складові і, водночас, ще й надати ім'я для всієї купи. Це досягається написанням імені і `@` як префіксу до взірця. Наприклад, взірець `xs@(x:y:ys)`. Цей взірець зіставиться точнісінько з тим самим, що й `x:y:ys`, але ви також одним маєте змогу отримати цілий список за допомогою `xs`, і не треба буде повторюватися, знову набираючи `x:y:ys` у тілі функції. Ось простий як двері приклад:

```
capital :: String -> String
capital ""         = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
ghci> capital "Dracula"
"The first letter of Dracula is D"
```

Зазвичай ми використовуємо взірці з іменами, щоб уникнути повторів, коли зіставляємо щось із взірцем, а у відповідному тому взірцеві тілі функції треба звертатись не тільки до того щось в цілому, а й також треба доступитися до його складових.

Ще одне — у зіставленні із взірцем не можна використовувати `++`. Якщо ви спробуєте зіставити `(xs ++ ys)` із взірцем, то що ж опиниться в першому, а що у другому списках? Це немає сенсу робити через неоднозначність. Але от має

сенс зіставити із `(xs ++ [x,y,z])` або просто `(xs ++ [x])`, але це неможливо зробити через однозв'язну природу списків в Хаскелі.

4.2 Варта, варта!

В той час як взірці перевіряють чи відповідає **вхід** [input] певній формі і деконструюють його згідно неї, вартові перевіряють, чи певна властивість (чи властивості) вхідних даних є правдивою чи хибною. Це дуже нагадує інструкцію розгалуження, і дійсно — вони дуже схожі. Справа в тому, що вартові легше читаються коли ми маємо кілька умов і вони особливо зручні при використанні в тандемі із взірцями.

Не буду пояснювати їхній синтаксис — краще спробуймо відразу написати функцію, використовуючи вартові. Напишемо просту функцію, яка вас похвалить чи, навпаки, насварить, залежно від того, який у вас індекс маси тіла (ІМТ). Щоб порахувати ІМТ, треба поділити свою вагу на зріст в квадраті. Якщо ваш ІМТ менший за 18.5, у вас недостатня маса тіла. Якщо він становить від 18.5 до 25, у вас нормальна вага. Від 25 до 30 — у вас надлишкова маса тіла, а понад 30 — ожиріння. Ось функція (ми не будемо зараз нічого обчислювати, ця функція просто отримує ІМТ і сварить вас):

```
bmiTell :: RealFloat a => a -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. " ++
                    "Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
```

Вартові позначаються вертикальними рисками (трубами), які ставляться після імені функції та її параметрів. Зазвичай їх трохи зсувають вправо і вишикують. Вартовий — це, по суті, булів вираз. Якщо результатом обрахунку вартового є `True`, тоді використовується відповідне тому вартовому тіло функції. Якщо `False` — контроль переходить до наступного вартового і так далі. Якщо подамо цій функції `24.3`, вона спершу перевірить, чи це число менше або дорівнює `18.5`. Оскільки це не так, перевірка переходить до наступного вартового. Другий вартовий перевіряє число, і оскільки `24.3` менше, ніж `25.0`, другий вартовий «пропускає» і повертається другий рядок.

Це дуже схоже на велике дерево «якщо-тоді-інакше» в імперативних мовах, тільки воно набагато краще і зручніше до відчитання. Великі дерева «якщо-

тоді-інакше» не бажано використовувати, але часом задача означена так, що без дерева ніяк не обійдешся. Вартові — чудова альтернатива таким деревам.

Дуже часто останній вартовий — це `otherwise`. `otherwise` означене просто як `otherwise = True` і він вловлює все. Схоже на взірці, але ті лише перевіряють, чи вхідні дані узгоджуються із ними, тоді як вартові здійснюють перевірку булевих умов. Якщо всі вартові функції після обрахування дорівнюватимуть `False` (і ми не дописали універсального вартового `otherwise`), оцінювання переходить до наступного **взірця**. Ось як взірці та вартові чудово співпрацюють. Якщо не знайдено відповідного вартового чи взірця, буде викинуто помилку.

Звісно, що вартових можна використовувати з функціями, які приймають стільки параметрів, скільки нам треба. Щоб не примушувати користувача самотужки рахувати свій ІМТ перед тим, як викликати функцію, спробуймо змінити цю функцію так, щоб вона брала зріст і вагу і сама виконувала всі обрахунки.

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. " ++
                                    "Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! " ++
                                    "Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Подивимось, чи я товстий...

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
```

Ура! Я не товстий! Але Хаскел сказав, що я бридкий! Ну й хай собі!

Зверніть увагу, що між параметрами функції і першим вартовим не ставиться `=`. Чимало новачків отримують синтаксичні помилки бо вони туди його тулять.

Ще один простенький приклад: реалізуймо свою власну функцію `max`. Як ви пригадуєте, вона бере два значення, які можна порівнювати, і повертає більше з них.

```
max' :: Ord a => a -> a -> a
max' a b
    | a > b    = a
    | otherwise = b
```

Вартових можна писати одним рядком тексту також, але краще так не робити, бо тоді код важче читатиметься, навіть якщо функції коротенькі. Проте для

прикладу можемо записати `max'` ось так:

```
max' :: Ord a => a -> a -> a
max' a b | a > b = a | otherwise = b
```

Бррр! Прочитати взагалі неможливо! Йдемо далі: реалізуймо свою власну `compare`, використовуючи вартових.

```
myCompare :: Ord a => a -> a -> Ordering
a `myCompare` b
    | a > b     = GT
    | a == b    = EQ
    | otherwise  = LT
```

```
ghci> 3 `myCompare` 2
GT
```

Примітка: Ми можемо не тільки викликати функції інфіксно (за допомогою спадного наголосу), а й означувати їх інфіксно також. Такі означення часом легше читаються.

4.3 Де!?

У попередньому розділі ми означили функцію, яка рахує ІМТ і висловлює своє «фе», ось так:

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. " ++
                                    "Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! " ++
                                    "Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

Зверніть увагу, що ми маємо три повтори. Ми повторюємося тричі. У програмуванні повторитися (тричі) — це так само добре, як отримати по голові. Оскільки ми повторюємо той самий вираз тричі, було б чудово, якби ми могли порахувати його один раз, прив'язати до імені, а тоді використати ім'я замість виразу. Можемо змінити нашу функцію ось так:

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
```

```
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. " ++
                    "Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

Ми пишемо ключове слово `where` після вартових (і перед ним найліпше давати такий самий відступ, як перед трубами), а тоді означуємо кілька змінних чи функцій. Ці імена видно усім вартовим, і ми більше не мусимо повторюватися. Якщо ми захочемо порахувати ІМТ по-іншому, нам треба внести зміни тільки один раз. По-друге, така річ як `weight / height ^ 2` тепер має ім'я `bmi`, і наш код стає легше читати. По-третє, наша програма працюватиме швидше (потенційно), адже `bmi` буде пораховано всього лиш один раз. А можемо трохи переборщити і записати цю функцію ось так:

```
bmiTell :: RealFloat a => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. " ++
                      "Pffft, I bet you're ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Імена з блоку `where` видимі тільки у функції до якої цей блок належить, і тому вони не засмітять простору імен інших функцій. Зверніть увагу, що всі імена вирівняні в одну колонку. Якщо ми їх гарненько і правильно не вирівняємо, Хаскел заплутається, адже тоді він не знатиме, що всі вони належать до одного блоку.

Тіла функцій що відповідають різним взірцям не можуть звертатися до спільного блоку `where`. Якщо ви хочете, щоб кілька взірців однієї функції мали доступ до якогось спільного імені, його потрібно буде означити глобально.

В зв'язках з блоку `where` можна також **зіставляти із взірцем**! Ми б могли переписати блок `where` з нашої попередньої функції ось як:

```
...
    where bmi = weight / height ^ 2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

Напишімо ще одну досить тривіальну функцію, де ми надаємо ім'я і прізвище людини і отримуємо у відповідь її ініціали.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Ми б могли зіставити безпосередньо у параметрах функції (і, власне, то було б коротше і зрозуміліше), але я просто хочу показати, що це можна зробити і в зв'язках з `where`.

Ми означували сталі в блоках `where`, але там можна означувати також і функції. Не зраджуючи нашому здоровому способу програмування, створімо функцію, яка бере список пар вага-зріст і повертає список з кількох ІМТ.

```
calcBmis :: RealFloat a => [(a, a)] -> [a]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

І це все! У цьому прикладі ми означили функцію `bmi` тому що нам її треба буде викликати багато разів для кожної пари зі списку що подано як параметр. Ми мусимо опрацювати список що надходить, а там для кожної пари — різний ІМТ.

Зв'язки `where` бувають і вкладеними. Це поширена ідіома — означувати допоміжну функцію в секції `where` іншої функції. Ну а далі і в допоміжній функції можна означувати нові функції, кожна з яких може мати свою секцію `where`.

4.4 Let it be[†]

Зв'язки `let` дуже схожі на зв'язки `where`. Зв'язки `where` — це синтаксична конструкція, що дає вам змогу зробити прив'язку до змінних у кінці функції, і змінні буде видно в усьому тілі тієї функції, у тому числі і в вартових. А от зв'язки `let` дозволяють робити прив'язку до змінних будь-де. До того ж, вони самі є виразами (а не інструкціями). Оскільки зв'язки `let` є дуже локальними означення з `let` «не перебігають» з вартового до вартового. Як і будь-які конструкції в Хаскелі, які прив'язують значення до імен, зв'язки `let` добре працюють із зіставленням із взірцем. Перевіримо `let` на практиці! Ось як можна означити функцію, що вираховує площу поверхні циліндра з його висоти та радіуса:

[†] **«Хай буде так» — назва останнього студійного альбому гурту The Beatles.**

```
cylinder :: RealFloat a => a -> a -> a
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea  = pi * r ^ 2
    in  sideArea + 2 * topArea
```

Загальна форма отака: `let` <зв'язки> `in` <вираз>. Імена, що означені у частині що йде після `let`, доступні в виразі після `in`. Як бачите, ми могли б це означити і за допомогою блоку `where`. Зверніть увагу, що імена також вирівняні в одну колонку. Отож, у чому різниця між цими двома підходами? Наразі здається, що `let` спочатку подає зв'язки, а пізніше — вираз у якому вони використовуються, тоді як `where` робить навпаки.

Різниця полягає в тому, що зв'язки `let` є виразами. Зв'язки `where` є всього лиш синтаксичними конструкціями (інструкціями). Пригадуєте, коли ми говорили про інструкцію розгалуження, я пояснював, що інструкція «якщо-тоді-інакше» є також виразом, і тому її можна запхнути куди завгодно?[†]

```
ghci> [if 5>3 then "Woo" else "Boo", if 'a'>'b' then "Foo" else "Bar"]
["Woo", "Bar"]
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
```

Те ж саме можна зробити і зі зв'язками `let`.

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

`let` також використовують для означення функцій локально:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

Якщо ми хочемо означити кілька змінних в одному рядку тексту, ми, звичайно ж, не можемо вишикувати їх у колонку. Але ми можемо відокремлювати їх крапкою з комою.

```
ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey ";
bar = "there!" in foo ++ bar)
```

[†]Інструкції не повертають результати, а вирази — так.

```
(6000000,"Hey there!")
```

Після останньої зв'язки не обов'язково ставити крапку з комою, але якщо хочете — будь ласка. Як я вже казав, в зв'язках `let` можна зіставляти із взірцем. Це допоможе швидко розвалити кортеж на складові і прив'язати їх до імен. Отаке.

```
ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100
600
```

Зв'язки `let` можна вставляти всередину спискових характерів. Спробуймо переписати наш попередній приклад де обробляється список пар вага-зріст, використовуючи `let` всередині спискового виразу (а не означуючи допоміжну функцію за допомогою `where`, як це було раніше).

```
calcBmis :: RealFloat a => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

Ми вставляємо `let` всередину спискового характеру так само, як предикат, але він не фільтрує список, а лише прив'язує до імен. Імена, означені в `let` всередині спискового виразу є видимими у функції виводу (частина перед `|`), а також для всіх предикатів і секцій, що йдуть після цієї зв'язки. То ж ми могли б змусити нашу функцію повертати лише ІМТ товстунів:

```
calcBmis :: RealFloat a => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

Ми не можемо використати ім'я `bmi` у частині `(w, h) <- xs`, оскільки вона була означена перед зв'язкою `let`.

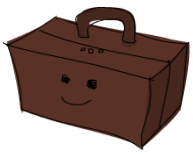
Ми не писали `in` у зв'язці `let`, коли використали її у списковому характері, оскільки в характерах видимість імен уже означено. Проте ми могли б використати зв'язку `let` із `in` у предикаті, і тоді означені там імена були б видимими тільки для того предиката. Означуючи функції та константи безпосередньо в GHCi, ми теж можемо вилучити частину `in`. У такому випадку імена будуть видимими під час усієї інтерактивної сесії.

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

Ви запитаєте: якщо зв'язки `let` такі круті, то чому б їх не використовувати постійно замість `where`? Справа в тім, що зв'язки `let` є виразами і тому доволі

локальні. Зокрема, вони не поширюються на вартових. Дехто надає перевагу зв'язкам `where`, тому що імена йдуть після функції, у якій вони використовуються. Тоді тіло функції ближче до її імені та оголошення її типу (в просторі коду), а такий код декому легше читати.

4.5 Вирази вибору



Чимало імперативних мов (C, C++, Java і так далі) мають конструкцію case, і якщо ви колись програмували такими мовами, то напевно знаєте, про що йдеться — про те, щоб взяти змінну, і виконувати різні блоки коду для різних її конкретних значень, із можливістю долучення універсального блоку коду на випадок, якщо змінна приймає якесь значення, для якого ми не налаштували спеціального блоку коду для обробки.

Хаскел запозичує це поняття і вдосконалює його. Як можемо здогадатися із назви, вирази вибору — це, нууууу, вирази, майже такі ж як вирази «якщо-тоді-інакше» і зв'язки `let`. Ми не тільки можемо обчислювати вирази, залежно від того, яке значення набуває змінна, а й зіставляти із взірцем. Хммм, взяти змінну, зіставити її із взірцем, порахувати щось залежно від її значення — де ми чули про це раніше? Ага! Зіставлення із взірцем в означеннях функцій! Насправді, такі означення є всього лише синтаксичним цукром для виразів вибору. Ці два шматки коду роблять те ж саме і є взаємозамінними:

```
head' :: [a] -> a
head' []    = error "No head for empty lists!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of []    -> error "No head for empty lists!"
                      (x:_) -> x
```

Бачите — вирази вибору мають простенький синтаксис:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

`expression` зіставляється із взірцем. Зіставляння із взірцем відбувається як завжди: повертається `result` для першого взірця який зіставляється з `expression`. Якщо провалюється повз увесь вираз вибору і відповідного взірця (і зіставлення) немає — отримуємо помилку виконання.

В той час як параметри функцій можна зіставити із взірцем лише під час означування функцій, вирази вибору можна використовувати де завгодно. До прикладу:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of
    []  -> "empty."
    [x] -> "a singleton list."
    xs  -> "a longer list."
```

Вирази вибору стануть в пригоді, коли ми зіставлятимемо щось із взірцем посередині виразу. Повторюючись, наголосимо, що зіставлення із взірцем в означеннях функцій — це синтаксичний цукор для виразів вибору, тому цю функцію можна було б означити і так:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what []  = "empty."
          what [x] = "a singleton list."
          what xs  = "a longer list."
```

Покажчик