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

2017-05-21T00:06:22Z
Версія v4.7-54-gda41cf2

Зміст

5 Рекурсія **1**
5.1 Привіт, рекурсіє! 1
5.2 Максимум крутизни 2
5.3 Ще трохи рекурсивних функцій 4
5.4 Швидко, відсортуйсь! 7
5.5 Думаючи рекурсивно 9

Покажчик **11**

Розділ 5

Рекурсія

Переклад українською Марини Стрельчук

5.1 Привіт, рекурсіє!

Ми познайомилися із рекурсією у попередньому розділі, а у цьому ми її розглянемо детальніше. Зокрема, поговоримо, про те, що таке рекурсія, чому вона важлива в Хаскелі і як можна отримувати компактні та елегантні розв'язки задач, якщо думати про них рекурсивно.

Якщо ви все ще не знаєте, що таке рекурсія, прочитайте це речення. Ха-ха! Жартую! Насправді, рекурсія — це спосіб означування функцій, де функція, яку ми означуємо, використовується всередині свого ж означення. В математиці означення часто даються рекурсивно. Наприклад, послідовність Фібоначчі означується рекурсивно. Спочатку ми означуємо два перших числа Фібоначчі нерекурсивно. Ми говоримо, що $F(0) = 0$ і $F(1) = 1$; це означає, що 0-ве і 1-ше числа Фібоначчі дорівнюють 0 і 1 відповідно. Для будь-якого іншого натурального числа, відповідне число Фібоначчі є сумою двох попередніх чисел Фібоначчі. Таким чином, $F(n) = F(n-1) + F(n-2)$. Отже, $F(3)$ рівне $F(2) + F(1)$, що дорівнює $(F(1) + F(0)) + F(1)$. Оскільки кінцевий вираз містить лише нерекурсивно означені числа Фібоначчі, ми можемо з упевненістю сказати, що $F(3)$ дорівнює 2. В означенні рекурсії наявність одного або двох елементів, які задані нерекурсивно (як от $F(0)$ та $F(1)$ з нашого прикладу), на-

зивають **граничною умовою**. Граничні умови є важливими і їх треба задавати, якщо ви хочете, щоб ваша рекурсивна функція мала змогу завершити свою роботу. Якби ми не означили $F(0)$ та $F(1)$ нерекурсивно, ми б ніколи не отримали відповіді, незалежно від того, яке б значення не подавали: досягнувши 0, ми б тоді просто перейшли до від'ємних чисел. Бац, і раптом вже вираховуємо $F(-2000)$, що є $F(-2001) + F(-2002)$, і, отже, мусимо продовжувати далі — а кінця розрахунків наразі не видно!

Рекурсія є важливою в Хаскелі, тому що, на відміну від імперативних мов, в Хаскелі ми робимо обчислення в спосіб задання *означення* того, **що** ми хочемо отримати, замість *описання*, **як** його треба отримувати. Саме тому в Хаскелі і немає циклів while та for, а замість них нам частенько доводиться використовувати рекурсію.

5.2 Максимум крутизни

Функція `maximum` бере список об'єктів, які можуть бути впорядковані (втілення типокласу `Ord`), і повертає найбільший із них. Подумайте, як ви б реалізували це на імперативний манер. Ви, напевно, створили б змінну, в якій зберігали б поточне максимальне значення, і після того в циклі порівнювали б його з кожним елементом списку. Якщо поточне максимальне значення менше за значення елементу, ви б замінили його на цей елемент. Максимальне значення, яке залишається в кінці, і є результатом. Хух! Доволі багато слів пішло на описання такого простого алгоритму!

Тепер розгляньмо рекурсивне означення. Для початку створімо граничну умову, де скажемо, що максимум одноелементного списку дорівнює значенню єдиного елементу цього списку. Далі кажемо, що максимум більш довгого списку дорівнює голові, якщо вона більша ніж максимум хвоста даного списку. Якщо ж максимальне значення хвоста більше за голову, тоді це значення і є максимумом списку. Ось і все! Тепер реалізуймо це на Хаскелі.

```
maximum' :: Ord a => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

Як бачимо, зіставлення із взірцем чудово поєднується із рекурсією! Більшість імперативних мов не використовують зіставлення із взірцем, тому необхідно вживати інструкції розгалуження для перевірки граничних умов. Ну а

ми просто використовуємо тут взірці. Таким чином, перша гранична умова каже: якщо список пустий, завершуємося аварійно! Таке твердження є логічним, адже що є максимумом порожнього списку? Уявлення не маю. Другий взірець є реалізацією другої граничної умови. Якщо це одноелементний список, ми просто повертаємо єдиний елемент цього списку.

Третє означення виконує всю роботу. Ми зіставляємо із взірцем для розвалення списку на голову та хвіст. В рекурсивній обробці списків — це дуже поширена ідіома, тому звикайте до неї. Ми використовуємо зв'язку `where` для означування `maxTail` — покладаємо його рівним максимуму хвоста. Потім перевіряємо, чи значення голови більше ніж максимум хвоста. Якщо так, повертаємо голову списку. В іншому випадку — повертаємо максимум хвоста.

До прикладу, розгляньмо список чисел `[2,5,1]` і перевіримо роботу нашої функції. Якщо ми застосуємо `maximum'` до цього списку, перші два взірці не зіставляться. А от третій — так, і тому наш список розділиться на `2` і `[5,1]`. `where` захоче знайти максимум `[5,1]`, і тому виконання піде цим шляхом. `[5,1]` знову зіставиться лише із третім взірцем, і `[5,1]` буде поділено на `5` та `[1]`. Знову ж таки, зв'язка `where` захоче знайти максимум `[1]`. Тут вперше «вистрілює» гранична умова, і як максимум повертається `1`. Нарешті! Тепер ми робимо один крок нагору і порівнюємо `5` та максимум `[1]` (тобто, `1`), і ми звичайно ж отримуємо `5`. Отже, тепер ми знаємо, що максимум `[5,1]` дорівнює `5`. Йдемо вгору ще на один крок, де ми працювали із `2` і `[5,1]`. Порівнюючи `2` із максимумом `[5,1]` (який, як виявилося є `5`), ми вибираємо `5`.

А тепер іще більш зрозумілий варіант написання цієї функції — із застосуванням `max`. Якщо ви пам'ятаєте, `max` — це функція, яка приймає два значення і повертає більше з них. Ось як ми можемо переписати `maximum'` із використанням `max`:

```
maximum' :: Ord a => [a] -> a
maximum' []     = error "maximum of empty list"
maximum' [x]    = x
maximum' (x:xs) = max x (maximum' xs)
```

Елегантно, чи не так?! По суті, максимум списку — це `max` першого елементу і `maximum'` хвоста.

5.3 Ще трохи рекурсивних функцій

Тепер, коли ми знаємо, як думати рекурсивно, напишімо декілька функцій із використанням рекурсії. Перш за все, ми реалізуємо `replicate`. `replicate` приймає `Int` і деякий елемент, та повертає список, що містить кілька повторень того елементу. Наприклад, `replicate 3 5` повертає `[5,5,5]`. Подумаймо про граничні умови. Гадаю, що гранична умова то є нуль або якесь значення менше за нуль. Якщо ми спробуємо повторити щось нуль разів, ми маємо отримати порожній список. Це виконується також і для від'ємних чисел, тому що копіювати елемент від'ємну кількість разів не має сенсу.

```
replicate' :: (Num i, Ord i) => i -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x:replicate' (n-1) x
```

Тут ми використовуємо вартових замість взірців, оскільки ми перевіряємо булеву умову. Якщо `n` менше або рівне 0, повертаємо порожній список. В іншому випадку повертаємо список, де `x` стоїть в голові, а хвостом є список, де `x` повторене n-1 разів. Зрештою, `(n-1)` частина зробить виклик, в якому спрацює гранична умова.

Примітка: `Num` не є підкласом `Ord`. Це означає, що сутність, яка є числом, не обов'язково повинна мати впорядкування. Ось чому ми маємо накладати дві умови типокласів, — `Num` та `Ord`, коли додаємо чи віднімаємо і, водночас, порівнюємо.

Йдемо далі — реалізуємо `take`. `take` бере якусь кількість елементів з початку списку. Наприклад, `take 3 [5,4,3,2,1]` поверне `[5,4,3]`. Якщо ми спробуємо взяти 0 або меншу кількість елементів зі списку, отримаємо порожній список. Крім того, якщо ми спробуємо взяти будь-що з порожнього списку, ми отримаємо порожній список. Зверніть увагу — маємо дві граничні умови, навіть не напружившись. Розпишімо їх:

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Перший взірець стверджує: якщо ми спробуємо взяти 0 або від'ємне число елементів, отримаємо порожній список. Зверніть увагу, що ми використовуємо `_` для зіставлення зі списком — тому що нам байдуже *що* то є. Також зауважте, що ми використовуємо вартового, але без `otherwise` частини. Це означає, що коли `n` є більшим за 0, зіставляння «провалиться» до наступного взірця. Другий взірець каже: при спробі взяти щось із порожнього списку ми отримаємо порожній список. Третій взірець розбиває список на голову і хвіст. Тоді ми означуємо, що `n` перших елементів зі списку `(x:xs)` то є список, де `x` є головою, а хвіст складається з результату взяття `n-1` елементів з хвоста списку на вході — себто з `xs`. Спробуйте розписати на аркуші паперу, як виглядатиме процес обчислення результату, якщо ми спробуємо взяти, скажімо, 3 з `[4,3,2,1]`.

`reverse` просто розвертає список. Подумайте про граничні умови. Які во-

ни? Що там думати — це порожній список! Розвернений порожній список дорівнює порожньому спискові (самому собі). Гаразд. А як щодо решти варіантів? Ну, ви можете сказати, що, розділивши список на голову і хвіст, розвернення списку дорівнюватиме розверненню хвоста із додаванням голови в кінці.

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Ось так!

Оскільки Хаскел підтримує нескінченні списки, наша рекурсія не обов'язково повинна мати граничну умову. Проте, якщо граничних умов немає, рекурсія буде або ковбасити щось нескінченно довго, або побудує нескінченну структуру даних, як, наприклад, нескінченний список. Однак, приємність роботи із нескінченними списками полягає в тому, що ми можемо обрізати їх там, де забажаємо. `repeat` приймає елемент і повертає нескінченний список, який містить тільки цей елемент. Рекурсивна реалізація є доволі простою, — дивіться:

```
repeat' :: a -> [a]
repeat' x = x:repeat' x
```

Виклик `repeat 3` поверне список, який починається з `3` та містить нескінченну кількість трійок у хвості. Тому виклик `repeat 3` буде пораховано як `3:repeat 3`, що є `3:(3:repeat 3)`, що, в свою чергу, є `3:(3:(3:repeat 3))`, і так далі. `repeat 3` ніколи не завершить роботу, проте — `take 5 (repeat 3)` поверне список з п'яти трійок. По суті, це те саме що і `replicate 5 3`.

`zip` бере два списки і «застібає» їх разом отак: `zip [1,2,3] [2,3]` повертає `[(1,2),(2,3)]`. Якщо довжина списків на вході різна, `zip` обрізає довший список до довжини коротшого. Що ми отримаємо, `zip`-нувши щось із порожнім списком? Порожній список. Це і буде нашою граничною умовою. Проте `zip` отримує як параметри два списки, тому насправді існує дві граничні умови:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y):zip' xs ys
```

Перші два взірці кажуть, що, якщо перший або другий список порожні, ми отримаємо порожній список. Згідно з третім взірцем, два списки дорівнюють паруванню голів цих списків та застібанню їх хвостів. Застібання `[1,2,3]` та `['a','b']` в якийсь момент спробує поєднати `[3]` з `[]`. Тоді спрацю-

ють граничні умови і тому ми отримаємо `(1,'a'):(2,'b'):[]`, що є тотожнім `[(1,'a'),(2,'b')]`.

Реалізуймо ще одну стандартну функцію — `elem`. Ця функція приймає деякий елемент і список та перевіряє, чи містить даний список цей елемент. Граничною умовою, як і в більшості випадків при роботі зі списками, є порожній список. Ми знаємо, що порожній список (чисел, звірів, непрочитаних книжок і так далі.) не містить елементів взагалі, тому ми гарантовано не знайдемо там нічого, *що* б ми не шукали.

```
elem' :: Eq a => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

Досить просто і очікувано. Якщо голова списку не дорівнює нашому елементові, то ми перевіряємо хвіст. Якщо ми досягнемо порожнього списку, то результатом буде `False`.

5.4 Швидко, відсортуйсь!

У нас є список з елементів, які можуть бути відсортовані. Їх тип є втіленням типокласу `Ord`. І тепер, ми хочемо відсортувати їх! Існує дуже класний алгоритм сортування, який називається швидким сортуванням. Це доволі нетривіальний спосіб сортування елементів. Імперативні реалізації зазвичай потребують більше десяти рядків коду, а от на Хаскелі реалізація є елегантною та набагато коротшою. Алгоритм швидкого сортування став своєрідною візитною карткою Хаскела. Тому реалізуймо цей алгоритм і тут, хоча, варто зазначити, що це заняття є дещо показовим, а реалізація стала певним кліше, адже всі її використовують для демонстрації елегантності Хаскела.

Отже, почнімо з сиґнатури типу — маємо `quicksort :: Ord a => [a] -> [a]`. Жодних сюрпризів! Граничні умови? Як і очікувалося — порожній список. Відсортований порожній список — це порожній список. «На пальцях» алгоритм виглядає отак: **відсортований список є списком, де спочатку ідуть елементи менші (або рівні) за голову списку на вході (і ці елементи є відсортованими), потім власне голова, а потім — елементи, які за значенням більші за голову (ці елементи теж є відсортовани-**

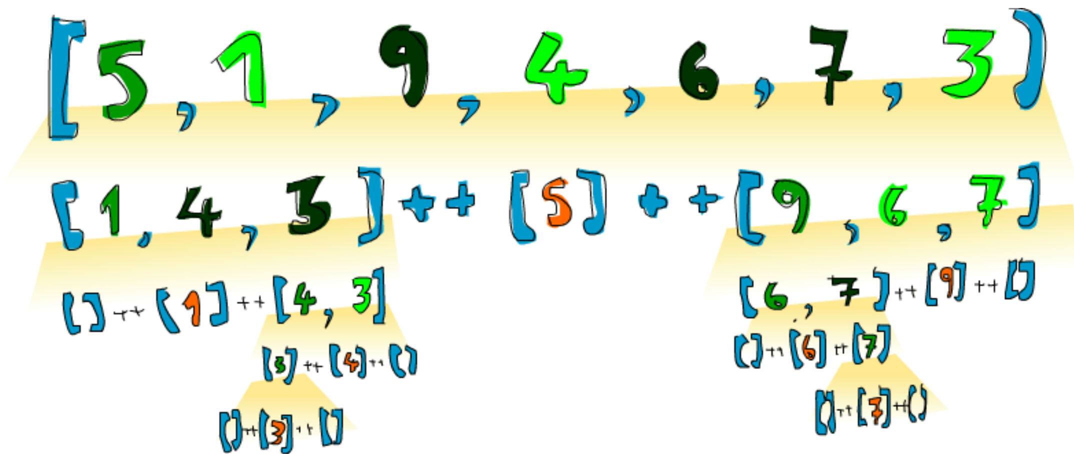
ми). Зверніть увагу, ми використали «є відсортованими» двічі в цьому означенні, тому швидше за все доведеться рекурсивно викликати себе двічі! Зверніть також увагу, що у означенні нашого алгоритму ми використали дієслово *є* замість імперативної мантри *виконайте спочатку це, потім оце, і згодом зробіть оте...* В цьому і є краса функційного програмування! Як будемо фільтрувати список, щоб отримати лише елементи менші за голову, та елементи, які є більші за голову? Спискові характери. Отож, до справи — ось як виглядатиме означення цієї функції:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted  = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Трохи потестуймо наш алгоритм, щоб переконатися в його коректності.

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

Нарешті! Те що треба! Скажімо ми хочемо відсортувати `[5,1,9,4,6,7,3]`. Цей алгоритм спочатку візьме голову, яка є `5`, і розташує її всередині двох списків. До першого потрапляють елементи менші за голову (або такі самі як голова), а до другого — більші. Отже, у певний момент матимемо `[1,4,3] ++ [5] ++ [9,6,7]`. Тепер відомо, що у повністю відсортованому списку `5` стоятиме на четвертому місці, оскільки ми маємо 3 числа менші за 5 та 3 числа більші за 5. Далі, якщо ми відсортуємо `[1,4,3]` і `[9,6,7]`, ми отримаємо повністю відсортований список! Ми сортуємо ці два підсписки, використовуючи цю ж саму функцію. Таким чином, ми розбиватимемо підсписки на дедалі коротші підпідсписки аж доки не отримаємо порожні списки, а порожні списки вже й так відсортовані. Ось ілюстрація:



Елементи, які потрапили на свої місця і не будуть змінювати позиції, позначено **помаранчевим кольором**. Якщо ви прочитаєте їх зліва направо, ви побачите відсортований список. Хоча ми вирішили порівнювати елементи з хвостів із головами, ми могли б використовувати будь-який елемент замість голови у порівнянні. В швидкому сортуванні, елемент, із яким порівнюються решта елементів, називається стрижнем. Стрижні тут позначено **зеленим**. Ми вибрали голову, тому що її легко отримати, використовуючи зіставлення із взірцем. Елементи, які є менші за стрижень — **світло-зелені**, а елементи більші за стрижень — **темно-зелені**. Градієнтні жовтуватощі тут позначають застосування швидкого сортування.

5.5 Думаючи рекурсивно

Ми доволі добре попрацювали із рекурсією, і як ви, напевно, помітили, тут є певна ідіома. Зазвичай ви спочатку означуєте граничний випадок, а потім — функцію, яка щось робить з якимось елементом та результатом застосування цієї функції до решти елементів. Не має значення, *що* це — список, дерево або будь-яка інша структура даних. Сума то є перший елемент списку плюс сума решти списку. Добуток елементів списку — перший елемент списку, помножений на добуток решти елементів списку. Довжина списку то є одиниця плюс довжина хвоста списку. І так далі, і так далі...

Звісно, приклади, наведені вище, всі мають якісь граничні випадки. Зазвичай, граничні випадки — це деякі сценарії, в яких застосування рекурсії не має сенсу. При роботі із списками граничним випадком найчастіше є порожній список. Якщо ви маєте справу з деревами, граничний випа-

док — це зазвичай вузол, який не має дітей.

Схожа ситуація і при рекурсивній роботі із числами. Як правило, йдеться про деяке число і функцію, аргументом якої є це число, але трохи модифіковане. Ми працювали із функцію факторіалу, яку було означено як добуток числа та факторіалу цього числа мінус один. Таке рекурсивне означення не має сенсу, якщо число дорівнює нулю, оскільки факторіал існує лише для додатних чисел. Досить часто значенням з граничного випадку виявляється нейтральний елемент. Нейтральним елементом для операції множення є 1, оскільки, помноживши будь-що на 1, ви отримаєте те будь-що без змін. Також, коли ми обчислюємо суму списків, ми означуємо суму порожнього списку як 0, і 0 тут — це нейтральний елемент операції додавання. В алгоритмі швидкого сортування граничним випадком є порожній список й нейтральним елементом також є порожній список, тому що при додаванні порожнього списку до будь-якого списку, ви просто отримаєте назад початковий список без змін.

Підсумовуючи: аби знайти рекурсивний розв'язок задачі, подумайте спочатку про випадки, де рекурсія не має сенсу, і спробуйте використати їх як граничні умови; потім розгляньте нейтральні елементи; наостанок, подумайте, як краще розбити параметри функції (наприклад, списки-параметри зазвичай розбиваються на голову і хвіст за допомогою зіставлення із взірцем) і до якої частини того розбиття ви будете застосовувати рекурсивний виклик.

Покажчик

childern of a node
: діти вузла, 10

edge condition
: гранична умова; крайова умова, 2

factorial function
: функція факторіалу, 10

fibonacci sequence
: послідовність Фібоначчі, 1

function definition
: означення функції, 1

identity element
: нейтральний елемент, 10

node of a tree
: вузол дерева, 10

non-recursive
: нерекурсивна, 1

pattern (common programming idiom)
: ідіома (поширений прийом чи розв'язок в програмуванні), 9

pivot
: стрижень, 9

quicksort algorithm
: алгоритм швидкого сортування, 10

recursion
: рекурсія, 1

reverse (e.g., of a list)
: розвернення (напр., списку), 6

singleton list
: односписок; одноелементний список, 3

typeclass constraint
: умова типокласу, 4

алгоритм швидкого сортування
quicksort algorithm, 10
вузол дерева
node of a tree, 10
гранична умова; крайова умова
edge condition, 2
діти вузла
childern of a node, 10
нейтральний елемент
identity element, 10
нерекурсивна
non-recursive, 1
односписок; одноелементний список
singleton list, 3
означення функції
function definition, 1
послідовність Фібоначчі
fibonacci sequence, 1
рекурсія
recursion, 1
розвернення (напр., списку)
reverse (e.g., of a list), 6
стрижень
pivot, 9
умова типокласу
typeclass constraint, 4
функція факторіалу
factorial function, 10
ідіома (поширений прийом чи розв'язок в програмуванні)
pattern (common programming idiom), 9